Effective Kafka

A **Hands-on Guide** to Building Robust and Scalable Event-Driven Applications with Code Examples in Java

Emil Koutanov

Effective Kafka

A Hands-On Guide to Building Robust and Scalable Event-Driven Applications with Code Examples in Java

Emil Koutanov

This book is for sale at http://leanpub.com/effectivekafka

This version was published on 2021-01-05

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Emil Koutanov by spreading the word about this book on Twitter!

The suggested hashtag for this book is #ApacheKafka.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#ApacheKafka

Dedicated to my family, who have unrelentlessly supported me as I disappeared for countless hours in my study, producing reams of mildly cohesive text.

Contents

Chapter 1: Event Streaming Fundamentals

It is amazing how the software engineering landscape has transformed over the last decade. Not long ago, applications were largely monolithic in nature, internally-layered, typically hosted within application servers and backed by 'big iron' relational databases with hundreds or thousands of interrelated tables. Distributed applications were the 'gold standard' by those measures — coarse-grained deployable units scattered among a static cluster of application servers, hosted on a fleet of virtual machines and communicating over SOAP-based APIs or message queues. Containerisation, cloud computing, elasticity, ephemeral computing, functions-as-a-service, immutable infrastructure — all niche concepts that were just starting to surface, making minor, barely perceptible ripples in an architectural institution that was otherwise well-set in its ways.

That was then. Today, these concepts are profoundly commonplace. Engineers are often heard interleaving several such terms in the same sentence; it would seem that the engineering community had miraculously stumbled upon an elixir that has all but cured us of our prior burdens — at least when it comes to developer velocity, time-to-market, system availability, scalability, and just about every other material concern that had kept the engineering manager of yore awake at night. Today, we have microservices in the cloud. Problem solved. Next question.

Except no such event *actually* occurred. We did not discover a solution to the problem; we merely shifted the problem. Aspects of software development that used to be straightforward in the 'old world', such as debugging, profiling, performance management, and state consistency — are now an order of magnitude more complex. On top of this, a microservices architecture brings its own unique woes. Services are more fluid and elastic, and tracking of their instances, their versions and dependencies is a Herculean challenge that balloons in complexity as the component landscape evolves. To top this off, services will fail in isolation, further exacerbated by unreliable networks, potentially leaving some activities in a state of partial completeness. Given a large enough system, parts of it may be suffering a minor outage at any given point in time, potentially impacting a subset of users, quite often without the operator's awareness.

With so many 'moving parts', how does one stay on top of these challenges? How does one make the engineering process *sustainable*? Or should we just write off the metamorphosis of the recent decade as a failed experiment?

The real challenges of distributed systems

If there is one thing to be learned from the opening gambit, it is that there is no 'silver bullet'. Architectural paradigms are somewhat like design patterns, but broader scoped, more subjective,

and far less prescriptive. However fashionable and blogged-about these paradigms might be, they only offer partial solutions to common problems. One must be mindful of the context at all times, and apply the model judiciously. And crucially, one must understand the deficiencies of the proposed approach, being able to reason about the implications of its adoption — both immediate and long-term.

The principal inconvenience of a distributed system is that it shifts the complexity from the innards of a service implementation to the notional fabric that spans across services. Some might say, it lifts the complexity from the *micro* level to the *macro* level. In doing so, it does not reduce the net complexity; on the contrary, it increases the aggregate complexity of the combined solution. An astute engineering leader is well-aware of this. The reason why a distributed architecture is often chosen — assuming it is chosen correctly — is to enable the compartmentalisation of the problem domain. It can, if necessary, be decomposed into smaller chunks and solved in partial isolation, typically by different teams — then progressively integrated into a complete whole. In some cases, this decomposition is deliberate, where teams are organised around the problem. In other, less-than-ideal cases, the breakdown of the problem is a reflection of Conway's Law, conforming to organisational structures. Presumed is the role an architect, or a senior engineering figure that orchestrates the decomposition, assuming the responsibility for ensuring the conceptual integrity and efficacy of the overall solution. Centralised coordination may not always be present — some organisations have opted for a more democratic style, whereby teams act in concert to decompose the problem organically, with little outside influence.

Coupling

Whichever the style of decomposition, the notion of *macro* complexity cannot be escaped. Fundamentally, components must communicate in one manner or another, and therein lies the problem: components are often inadvertently made aware of each other. This is called *coupling* — the degree of interdependence between software components. The lower the coupling, the greater the propensity of the system to evolve to meet new requirements, performance demands, and operational challenges. Conversely, tight coupling shackles the components of the system, increasing their mutual reliance and impeding their evolution.

There are known ways for alleviating the problem of coupling, such as the use of an asynchronous communication style and message-oriented middleware to segregate components. These techniques have been used to varying degrees of success; there are times where message-based communication has created a false economy — collaborating components may still be transitively dependent upon one another in spite of their designers' best efforts to forge opaque conduits between them.

Resilience

It would be rather nice if computers never failed and networks were reliable; as it happens, reality differs. The problem is exacerbated in a distributed context: the likelihood of any one component experiencing an isolated failure increases with the total number of components, which carries negative ramifications if components are interdependent.

Distributed systems typically require a different approach to resilience compared to their centralised counterparts. The quantity and makeup of failure scenarios is often much more daunting in distributed systems. Failures in centralised systems are mostly characterised as *fail-stop* scenarios — where a process fails totally and permanently, or a network partition occurs, which separates the entirety of the system from one or more clients, or the system from its dependencies. At either rate, the failure modes are trivially understood. By contrast, distributed systems introduce the concept of *partial failures*, *intermittent failures*, and, in the more extreme cases, *Byzantine failures*. The latter represents a special class of failures where processes submit incorrect or misleading information to unsuspecting peers.

Consistency

Ensuring state consistency in a distributed system is perhaps the most difficult aspect to get right. One can think of a distributed system as a vast state machine, with some elements of it being updated independently of others. There are varying levels of consistency, and different applications may demand specific forms of consistency to satisfy their requirements. The stronger the consistency level, the more synchronisation is necessary to maintain it. Synchronisation is generally regarded as a difficult problem; it is also expensive — requiring additional resources and impacting the performance of the system.

Cost being held a constant, the greater the requirement for consistency, the less distributed a system will be. There is also a natural counterbalance between consistency and availability, identified by Eric Brewer in 1998. The essence of it is in the following: distributed systems must be tolerant of network partitions, but in achieving this tolerance, they will have to either give up consistency or availability guarantees. Note, this conjecture does not claim that a consistent system cannot simultaneously be highly available, only that it must give up availability if a network partition does occur.

By comparison, centralised systems are not bound by the same laws, as they don't have to contend with network partitions. They can also take advantage of the underlying hardware, such as CPU cache lines and atomic operations, to ensure that individual threads within a process maintain consistency of shared data. When they do fail, they typically fail as a unit — losing any ephemeral state and leaving the persistent state as it was just before failure.

Event-Driven Architecture

Event-Driven Architecture (EDA) is a paradigm promoting the production, detection, consumption of, and reaction to *events*. An event is a significant state in change, that may be of interest within the domain where this state change occurred, or outside of that domain. Interested parties can be notified of an event by having the originating domain publish some canonical depiction of the event to a well-known conduit — a message broker, a ledger, or a shared datastore of some sort. Note, the event itself does not travel — only its notification; however, we often metonymically refer to the notification of the event as the event. (While formally incorrect, it is convenient.)

An event-driven system formally consists of *emitters* (also known as producers and agents), *consumers* (also known as subscribers and sinks), and *channels* (also known as brokers). We also use the term *upstream* — to refer to the elements prior to a given element in the emitter-consumer relation, and *downstream* — to refer to the subsequent elements.

An emitter of an event is not aware of any of the event's downstream consumers. This statement captures the essence of an event-driven architecture. An emitter does not even know whether a consumer exists; every transmission of an event is effectively a 'blind' broadcast. Likewise, consumers react to specific events without the knowledge of the particular emitter that published the event. A consumer need not be the final destination of the event; the event notification may be persisted or transformed by the consumer before being broadcast to the next stage in a notional pipeline. In other words, an event may spawn other events; elements in an event-driven architecture may combine the roles of emitters and consumers, simultaneously acting as both.

Event notifications are immutable. An element cannot modify an event's representation once it has been emitted, not even if it is the originally emitter. At most, it can emit new notifications relating to that event — enriching, refining, or superseding the original notification.

Coupling

Elements within EDA are exceedingly loosely coupled, to the point that they are largely unaware of one another. Emitters and consumers are only coupled to the intermediate channels, as well as to the representations of events — *schemas*. While some coupling invariably remains, in practice, EDA offers the lowest degree of coupling of any practical system. The collaborating components become largely autonomous, standalone systems that operate in their own right — each with their individual set of stakeholders, operational teams, and governance mechanisms.

By way of an example, an e-commerce system might emit events for each product purchase, detailing the time, product type, quantity, the identity of the customer, and so on. Downstream of the emitter, two systems — a business intelligence (BI) platform and an enterprise resource planning (ERP) platform — might react to the sales events and build their own sets of materialised views. (In effect, view-only projections of the emitter's state.) Each of these platforms are completely independent systems with their own stakeholders: the BI system satisfies the business reporting and analytics requirements for the marketing business unit, while the ERP system supports supply chain management and capacity planning — the remit of an entirely different business unit.

To put things into perspective, we shall consider the potential solutions to this problem in the absence of EDA. There are several ways one could have approached the solution; each approach commonly found in the industry to this day:

1. **Build a monolith**. Conceptually, the simplest approach, requiring a system to fulfill all requirements and cater to all stakeholders as an indivisible unit.
2. **Integration**. Allow the systems to invoke one another via some form of an API. Either the e-commerce platform could invoke the BI and ERP platforms at the point of sale, or the BI and ERP platforms could invoke the e-commerce platform APIs just before generating a business

report or supplier request. Some variations of this model use message queues for systems to send commands and queries to one another.

3. **Data decapsulation**. If system integrators were cowboys, this would be their prairie. Data decapsulation (a coined term, if one were to ask) sees systems 'reaching over' into each other's 'backyard', so to speak, to retrieve data directly from the source (for example, from an SQL database) — without asking the owner of the data, and oftentimes without their awareness.
4. **Shared data**. Build separate applications that share the same datastore. Each application is aware of all data, and can both read and modify any data element. Some variations of this scheme use database-level permissions to restrict access to the data based on an application's role, thereby binding the scope of each application.

Once laid out, the drawbacks of each model become apparent. The first approach — the proverbial monolith — suffers from uncontrolled complexity growth. In effect, it has to satisfy *everyone* and *everything*. This also makes it very difficult to change. From a reliability standpoint, it is the equivalent of putting all of one's eggs in one basket — if the monolith were to fail, it will impact all stakeholders simultaneously.

The second approach — integrate everything — is what these days is becoming more commonly known as the 'distributed monolith', especially when it is being discussed in the context of microservices. While the systems (or services, as the case may be) appear to be standalone — they might even be independently sourced and maintained — they are by no means autonomous, as they cannot change freely without impacting their peers.

The third approach — read others' data — is the architectural equivalent of a 'get rich quick scheme' that always ends in tears. It takes the path of least resistance, making it highly alluring. However, the model creates the tightest possible level of coupling, making it very difficult to change the parties down the track. It is also brittle — a minor and seemingly benign change to the internal data representation in one system could have a catastrophic effect on another system.

The final model — the use of a shared datastore — is a more civilised variation of the third approach. While it may be easier to govern, especially with the aid of database-level access control — the negative attributes are largely the same.

Now imagine that the business operates *multiple* disparate e-commerce platforms, located in different geographic regions or selling different sorts of products. And to top it off, the business now needs a separate data warehouse for long-term data collection and analysis. The addition of each new component significantly increases the complexity of the above solutions; in other words, they do not scale. By comparison, EDA scales perfectly linearly. Systems are unaware of one another and react to discrete events — the origin of an event is largely circumstantial. This level of autonomy permits the components to evolve rapidly in isolation, meeting new functional and non-functional requirements as necessary.

Resilience

The autonomy created by the use of EDA ensures that, as a whole, the system is less prone to outage if any of its individual components suffer a catastrophic failure. How is this achieved?

Integrated systems, and generally, any topological arrangement that exhibits a high degree of component coupling is prone to *correlated failure* — whereby the failure of one component can take down an entire system. In a tightly coupled system, components directly rely on one another to jointly achieve some goal. If one of these components fails, then the remaining components that depend on it may also cease to function; at minimum, they will not be able to carry out those operations that depend on the failed component.

In the case of a monolith, the failure assertion is trivial — if a fail-stop scenario occurs, the entire process is affected.

Under EDA, enduring a component failure implies the inability to either emit events or consume them. In the event of emitter failure, consumers may still operate freely, albeit without a facility for reacting to new events. Using our earlier example, if the e-commerce engine fails, none of the downstream processes will be affected — the business can still run analytical queries and attend to resource planning concerns. Conversely, if the ERP system fails, the business will still make sales; however, some products might not be placed on back-order in time, potentially leading to low stock levels. Furthermore, provided the event channel is durable, the e-commerce engine will continue to publish sales events, which will eventually be processed by the ERP system when it is restored. The failure of an event channel can be countered by implementing a local, stateful buffer on the emitter, so that any backlogged events can be published when the channel has been restored. In other words, not only is an event-driven system more resilient by retaining limited operational status during component failure, it is also capable of self-healing when failed components are replaced.

In practice, systems may suffer from soft failures, where components are saturated beyond their capacity to process requests, creating a cascading effect. In networking, this phenomenon is called 'congestive collapse'. In effect, components appear to be online, but are stressed — unable to turn around some fraction of requests within acceptable time frames. In turn, the requesting components — having detected a timeout — retransmit requests, hoping to eventually get a response. This increases pressure on the stressed components, exacerbating the situation. Often, the missed response is merely an indication of receiving the request — in effect, the requester is simply piling on duplicate work.

Under EDA, requesters do not require a confirmation from downstream consumers — a simple acknowledgement from the event channel is sufficient to assume that the event has been stably enqueued and that the consumer(s) will get to it at some future point in time.

Consistency

EDA ameliorates the problem of distributed consistency by attributing explicit mastership to state, such that any stateful element can only be manipulated by at most one system — its designated owner. This is also referred to as the originating domain of the event. Other domains may only react to the event; for example, they may reduce the event stream to a local projection of the emitter's state.

Under this model, consistency within the originating domain is trivially maintained by enforcing the single writer principle. External to the domain, the events can be replayed in the exact order

they were observed on the emitter, creating *sequential consistency* — a model of consistency where updates do not have to be seen instantaneously, but must be presented in the same order to all observers, which is also the order they were observed on the emitter. Alternatively, events may be emitted in *causal order*, categorising them into multiple related sequences, where events within any sequence are related amongst themselves, but unrelated to events in another sequence. This is a slight relaxation of sequential consistency to allow for safe parallelism, and is sufficient in the overwhelming majority of use cases.

Applicability

For all its outstanding benefits, EDA is not a panacea and cannot supplant integrated or monolithic systems in all cases. For instances, EDA is not well-suited to synchronous interactions, as mutual or unilateral awareness among collaborating parties runs contrary to the grain of EDA and negates most of its benefits.

EDA is not a general-purpose architectural paradigm. It is designed to be used in conjunction with other paradigms and design patterns, such as synchronous request-response style messaging, to solve more general problems. In the areas where it can be applied, it ordinarily leads to significant improvements in the system's non-functional characteristics. Therefore, one should seek to maximise opportunities for event-driven compositions, refactoring the architecture to that extent.

What is event streaming?

Finally, we arrive at the central question: *What is event streaming?* And frankly, there is little left to explain. There is but one shortfall in the earlier narrative: EDA is an architectural paradigm — it does not prescribe the particular semantics of the event interchange. Events could be broadcast among parties using different mechanisms, all potentially satisfying the basic tenets of EDA.

Event streaming is a mechanism that can be used to realise the *event channel* element in EDA. It is primarily concerned with the following aspects of event propagation:

- Interface between the emitter and the channel, and the consumer and the channel;
- Cardinality of the emitter and consumer elements that interact with a common channel;
- Delivery semantics;
- Enabling parallelism in the handling of event notifications;
- Persistence, durability, and retention of event records; and
- Ordering of events and associated consistency models.

The focal point of event streaming is, unsurprisingly, an *event stream.* At minimum, *an event stream is a durable, totally-ordered, unbounded sequence of immutable event records, delivered at least once to its subscriber(s).* An *event streaming platform* is a concrete technology that implements the event streaming model, addressing the points enumerated above. It interfaces with emitter and consumer

ecosystems, hosts event streams, and may provide additional functionality beyond the essential set of event streaming capabilities. For example, an event streaming platform may offer end-to-end compression and encryption of event records, which is not essential in the construction of event-driven systems, but is convenient nonetheless.

It is worth noting that event streaming is not required to implement the event channel element of EDA. Other transports, such as message queues, may be used to fulfill similar objectives. In fact, there is nothing to say that EDA is exclusive to distributed systems; the earliest forms of EDA were realised within the confines of a single process, using purely in-memory data structures. It may seem banal in comparison, but even UI frameworks of the bygone era, such as Java Swing, draw on the foundations of EDA, as do their more contemporary counterparts, such as React.

When operating in the context of a distributed system, the primary reason for choosing event streaming over the competing alternatives is that the former was designed specifically for use in EDA, and its various implementations — event streaming platforms — offer a host of capabilities that streamline their adoption in EDA. A well-designed event streaming platform provides direct correspondence with native EDA concepts. For example, it takes care of event immutability, record ordering, and supports multiple independent consumers — concepts that might not necessarily be endemic to alternate solutions, such as message queues.

This chapter has furnished an overview of the challenges of engineering distributed systems, contrasted with the building of monolithic business applications. The numerous drawbacks of distributed systems increase their cost and complicate their upkeep. Generally speaking, *the components of a complex system are distributed out of necessity* — namely, the requirement to scale in both the performance plane and in the engineering capacity to deliver change.

We looked at how the state of the art has progressed since the mass adoption of the principles of distributed computing in mainstream software engineering. Specifically, we explored *Event-Driven Architecture* as a highly effective paradigm for reducing coupling, bolstering resilience, and avoiding the complexities of maintaining a globally consistent state.

Finally, we touched upon *event streaming*, which is a rendition of the *event channel* element of EDA. We also learned why event streaming is the preferred approach for persisting and transporting event notifications. In no uncertain terms, event streaming is the most straightforward path for the construction of event-driven systems.

Chapter 2: Introducing Apache Kafka

Apache Kafka (or simply Kafka) is an event streaming platform. But it is also more than that. It is an entire ecosystem of technologies designed to assist in the construction of complete event-driven systems. Kafka goes above and beyond the essential set of event streaming capabilities, providing rich event persistence, transformation, and processing semantics.

Event streaming platforms are a comparatively recent paradigm within the broader message-oriented middleware class. There are only a handful of mainstream implementations available, compared to hundreds of MQ-style brokers, some going back to the 1980s (for example, Tuxedo). Compared to established messaging standards such as AMQP, MQTT, XMPP, and JMS, there are no equivalent standards in the streaming space. Kafka is a leader in the area of event streaming, and more broadly, event-driven architecture. While there is no *de jure* standard in event streaming, Kafka is the benchmark to which most competing products orient themselves. To this effect, several competitors — such as Azure Event Hubs and Apache Pulsar — offer APIs that mimic Kafka.

Event streaming platforms are an active area of continuous research and experimentation. In spite of this, event streaming platforms aren't just a niche concept or an academic idea with few esoteric use cases; they can be applied effectively to a broad range of messaging and eventing scenarios, routinely displacing their more traditional counterparts.

Kafka is written in Java, meaning it can run comfortably on most operating systems and hardware configurations. It can equally be deployed on bare metal, in the Cloud, and a Kubernetes cluster. And finally, Kafka has libraries written for just about every programming language, meaning that virtually every developer can start taking advantage of event streaming and push their application architecture to the next level of resilience and scalability.

The history of Kafka

Apache Kafka was originally developed by LinkedIn, and was subsequently open-sourced in early 2011. The name 'Kafka' was chosen by one of its founders — Jay Kreps. Kreps chose to name the software after the famous 20th-century author Franz Kafka because it was "a system optimised for writing". Kafka gained the full Apache Software Foundation project status in October 2012, having graduated from the Apache Incubator program.

Kafka was born out of a need to track and process large volumes of site events, such as page views and user actions, as well as for the aggregation log data. Before Kafka, LinkedIn maintained several disparate data pipelines, which presented a challenge from both complexity and operational scalability perspectives. In July 2011, having consolidated the individual platforms, Kafka was

processing approximately one billion events per day. By 2012, this number had risen to 20 billion. By July 2013, Kafka was carrying 200 billion events per day. Two years later, in 2015, Kafka was turning over one trillion events per day, with peaks of up to 4.5 million events per second.

Over the four years of 2011 to 2015, the volume of records has grown by three orders of magnitude. By the end of this period, LinkedIn was moving well over a petabyte of event data per week. By all means, this level of growth could not be attributed to Kafka alone; however, Kafka was undoubtedly a key enabler from an infrastructure perspective.

As of October 2019, LinkedIn maintains over 100 Kafka clusters, comprising more than 4,000 brokers. These collectively serve more than 100,000 topics and 7 million partitions. The total number of records handled by Kafka has surpassed 7 trillion per day.

The present day

The industry adoption of Kafka has been nothing short of phenomenal. The list of tech giants that heavily rely on Kafka is impressive in itself. To name just a few:

- **Yahoo** uses Kafka for real-time analytics, handling up to 20 gigabits of uncompressed event data per second in 2015. Yahoo is also a major contributor to the Kafka ecosystem, having open-sourced its in-house Cluster Manager for Apache Kafka (CMAK) product.
- **Twitter** heavily relies on Kafka for its mobile application performance management and analytics product, which has been clocked at five billion sessions per day in February 2015. Twitter processes this stream using a combination of Apache Storm, Hadoop, and AWS Elastic MapReduce.
- **Netflix** uses Kafka as the messaging backbone for its Keystone pipeline — a unified event publishing, collection, and routing infrastructure for both batch and stream processing. As of 2016, Keystone comprises over 4,000 brokers deployed entirely in the Cloud, which collectively handle more than 700 billion events per day.
- **Tumblr** relies on Kafka as an integral part of its event processing pipeline, capturing up 500 million page views a day back in 2012.
- **Square** uses Kafka as the underlying bus to facilitate stream processing, website activity tracking, metrics collection and monitoring, log aggregation, real-time analytics, and complex event processing.
- **Pinterest** employs Kafka for its real-time advertising platform, with 100 clusters comprising over 2,000 brokers deployed in AWS. Pinterest is turning over in excess of 800 billion events per day, peaking at 15 million per second.
- **Uber** is among the most prominent of Kafka adopters, processing in excess of a trillion events per day — mostly for data ingestion, event stream processing, database changelogs, log aggregation, and general-purpose publish-subscribe message exchanges. In addition, Uber is an avid open-source contributor — having released its in-house cluster replication solution *uReplicator* into the wild.

And it's not just the engineering-focused organisations that have adopted Kafka — by some estimates, up a third of Fortune 500 companies use Kafka to fulfill their event streaming and processing needs.

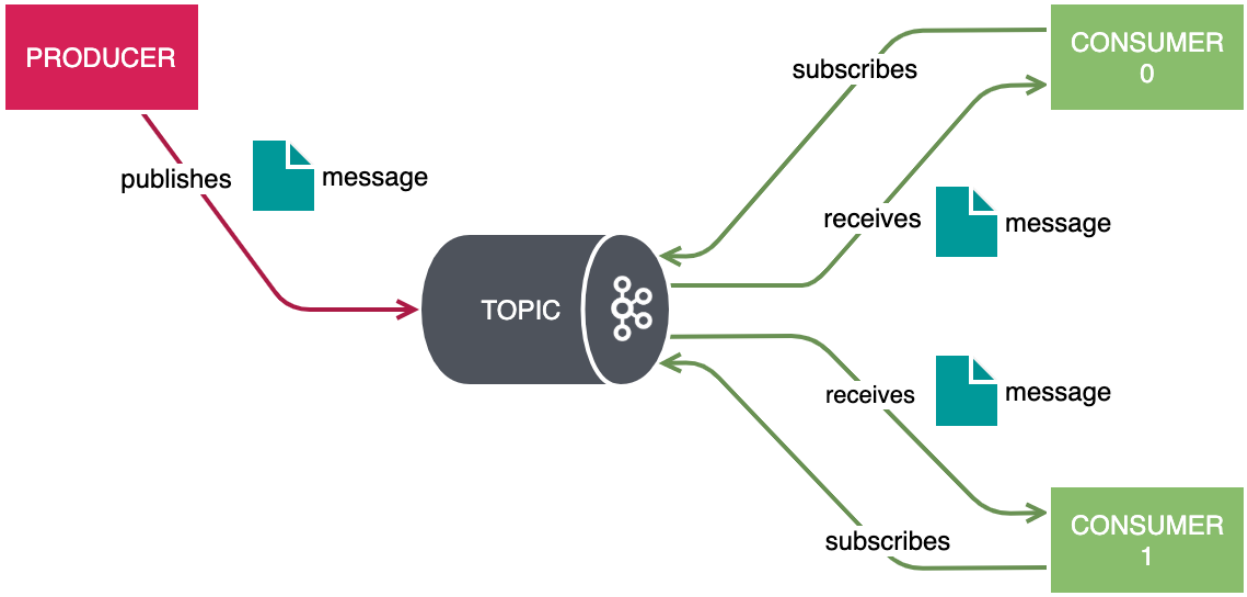
There are good reasons for this level of industry adoption. As it happens, Kafka is one of the most well-supported and well-regarded event streaming platforms, boasting an impressive number of open-source projects that integrate with Kafka. Some of the big names include Apache Storm, Apache Flink, Apache Hadoop, LogStash and the Elasticsearch Stack, to name a few There are also Kafka Connect integrations with every major SQL database, and most NoSQL ones too. At the time of writing, there are circa one hundred supported off-the-shelf connectors, which does not include custom connectors that have been independently developed.

Uses of Kafka

Chapter 1: Event Streaming Fundamentals has provided the necessary background, fitting Kafka as an event streaming platform within a larger event-driven system.

There are several use cases falling within the scope of EDA that are well-served by Apache Kafka. This section covers some of these scenarios, illustrating how Kafka may be used to address them.

Publish-subscribe



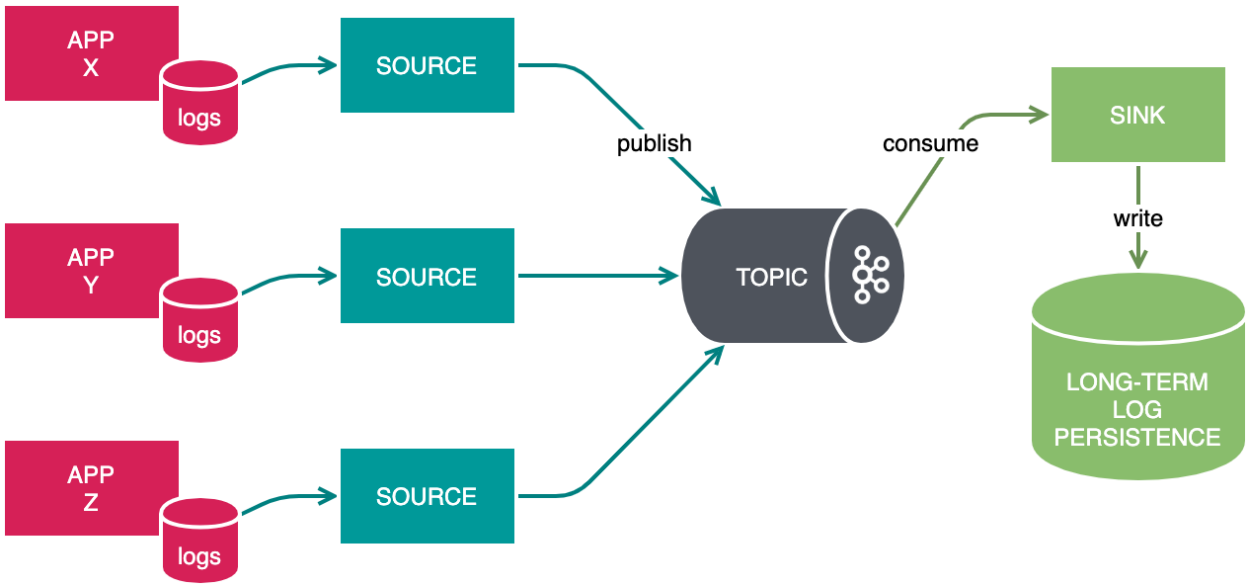
Pub-Sub

Any messaging scenario where producers are generally unaware of consumers, and instead publish messages to well-known aggregations called *topics*. Conversely, consumers are generally unaware of the producers but are instead concerned with specific content categories. The producer and consumer

ecosystems are loosely-coupled, being aware of only the common topic(s) and messaging schema(s). This pattern is commonly used in the construction of loosely-coupled microservices.

When Kafka is used for general-purpose publish-subscribe messaging, it will be competing with its 'enterprise' counterparts, such as message brokers and service buses. Admittedly, Kafka might not have all the features of some of these middleware platforms — such as message deletion, priority levels, producer flow control, distributed transactions, or dead-letter queues. On the other hand, these features are mostly representative of traditional messaging paradigms — intrinsic to how these platforms are commonly used. Kafka works in its own idiomatic way — optimised around unbounded sequences of immutable events. As long as a publish-subscribe relationship can be represented as such, then Kafka is fit for the task.

Log aggregation

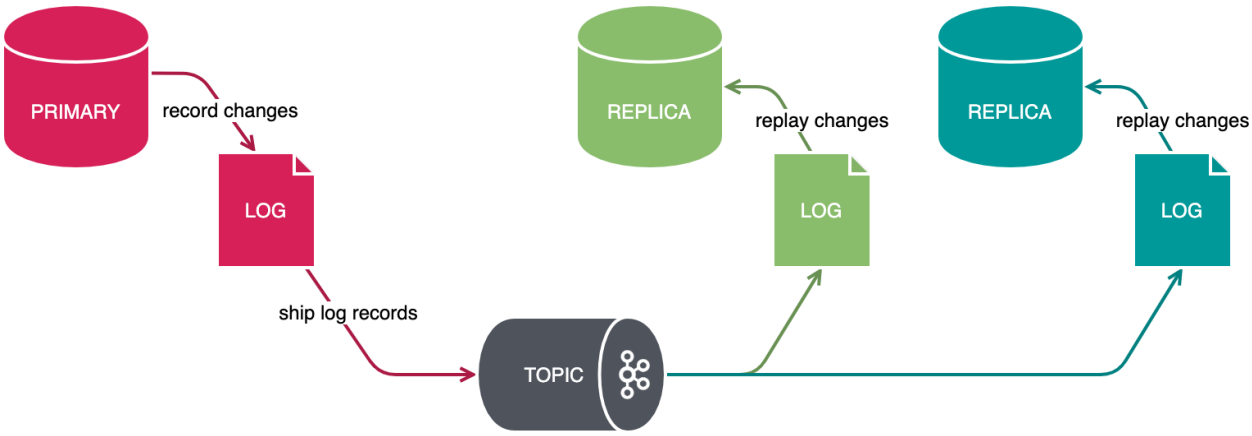


Log aggregation

Dealing with large volumes of log-structured events, typically emitted by application or infrastructure components. Logs may be generated at burst rates that significantly outstrip the ability of query-centric datastores to keep up with log ingestion and indexing, which are regarded as 'expensive' operations. Kafka can act as a buffer, offering an intermediate, durable datastore. The ingestion process will act as a sink, eventually collating the logs into a read-optimised database (for example, Elasticsearch or HBase).

A log aggregation pipeline may also contain intermediate steps, each adding value en route to the final destination; for example, to compress log data, encrypt log content, normalise the logs into a canonical form, or sanitise the log entries — scrubbing them of personally-identifiable information.

Log shipping



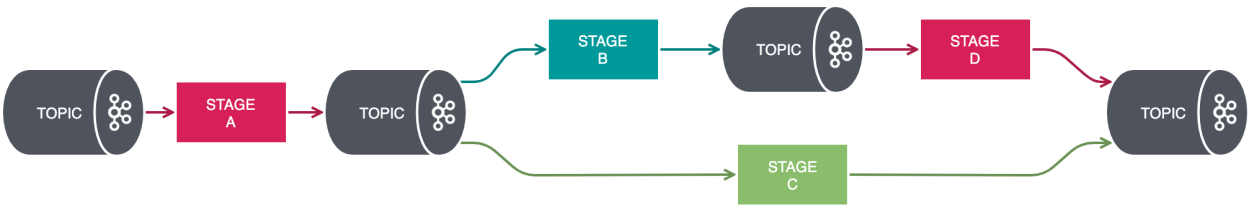
Log shipping

While sounding vaguely similar to log aggregation, the shipping of logs is a vastly different concept. Essentially, this involves the real-time copying of journal entries from a master data-centric system to one or more read-only replicas. Assuming stage changes are fully captured as journal records, replaying those records allows the replicas to accurately mimic the state of the master, albeit with some lag.

Kafka's optional ability to partition records within a topic to create independent, causally ordered sequences of events allows for replicas to operate in one of *sequential* or *causal* consistency models — depending on the chosen partitioning scheme. The various consistency models were briefly covered in Chapter 1: Event Streaming Fundamentals. Both consistency models are sufficient for creating read-only copies of the original data.

Log shipping is a key enabler for another related architectural pattern — *event sourcing*. Kafka will act as a durable event store, allowing any number of consumers to rebuild a point-in-time snapshot of their application state by replaying all records up to that point in time. Loss of state information in any of the downstream consumers can be recovered by replaying the events from the last stable checkpoint, thereby reducing the need to take frequent backups.

SEDA pipelines



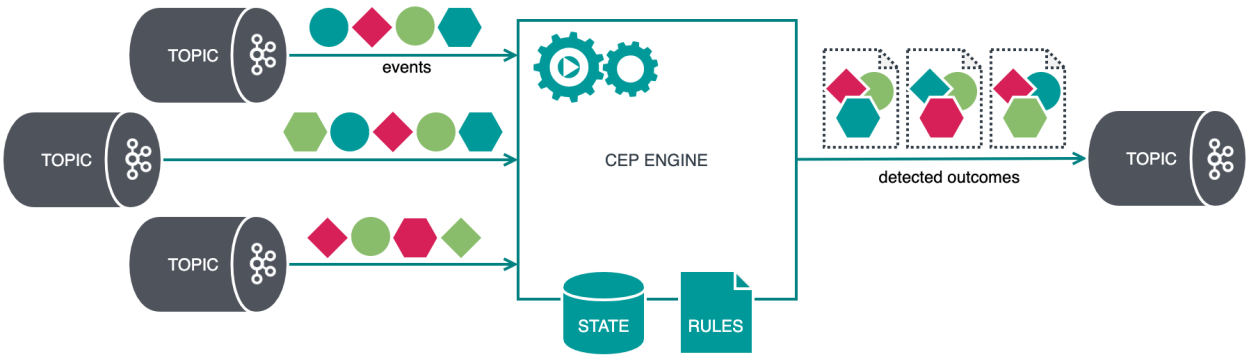
SEDA pipeline

Staged Event-Driven Architecture (SEDA) is the application of pipelining to event-oriented data. Events flow unidirectionally through a series of processing stages linked by topics, each one performing a mapping operation before publishing a transformed event to the next topic. Intermediate stages simultaneously act as both consumers and producers, and may scale autonomously and independently of one another to match their unique load demands. By breaking a complex problem into stages, SEDA improves the modularity of the system.

A SEDA pipeline may combine fan-in and fan-out topologies. Stages may consume events from multiple topics simultaneously, performing the equivalent of an SQL `JOIN` on event streams. Stages can also publish to multiple topics, feeding several downstream pipelines.

As a pattern, SEDA is readily found in data warehousing, data lakes, reporting, analytics, and other Business Intelligence systems, and is often a crucial element of Big Data applications. SEDA can also be used in log aggregation; in fact, log aggregation is a narrow specialisation of SEDA.

CEP

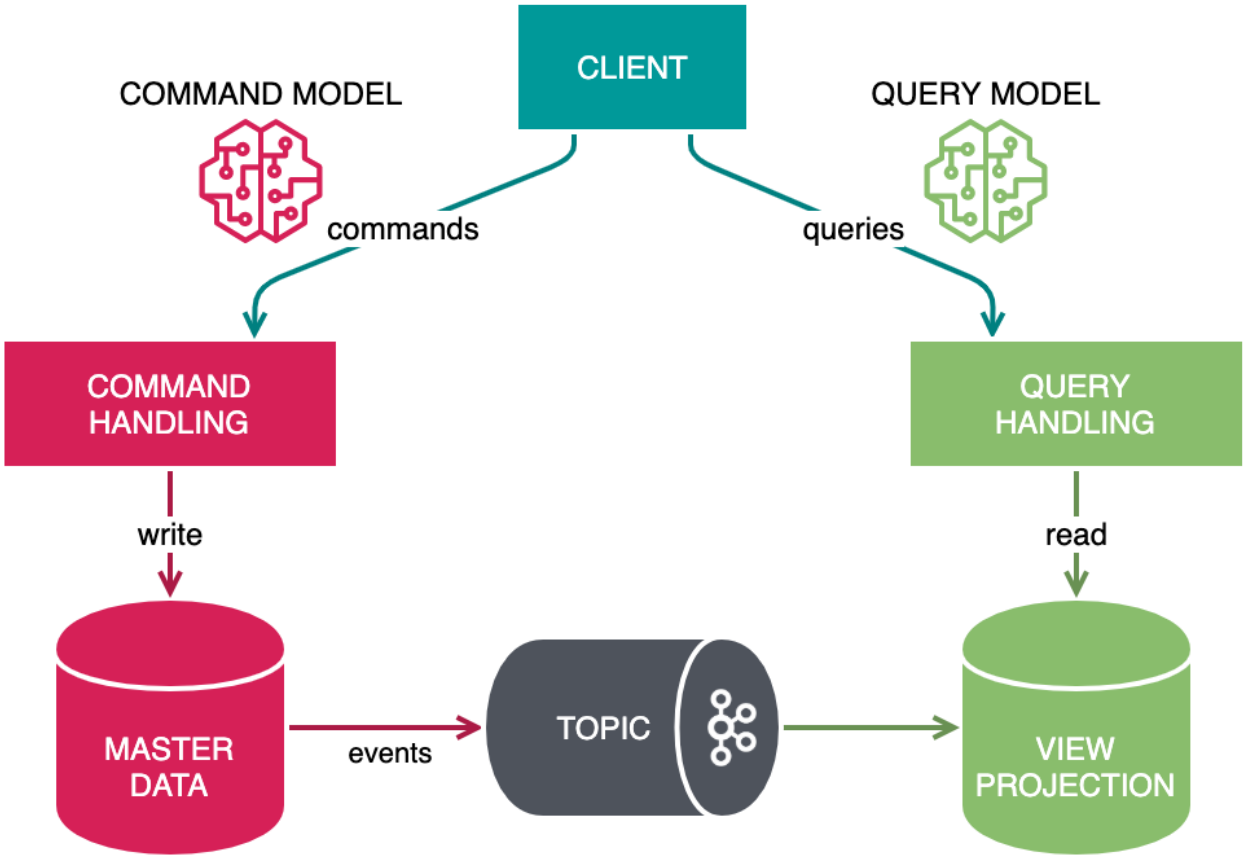


CEP

Complex Event Processing (CEP) extracts meaningful information and patterns in a stream of discrete events, or across a set of disjoint event streams. CEP processors tend to be stateful, as they must be able to efficiently recall prior events to identify patterns that might span a broad timeframe, ranging from milliseconds to days, depending on the context.

CEP is heavily employed in such applications as algorithmic stock trading, security threat analysis, real-time fraud detection, and control systems.

Event-sourced CQRS



CQRS

Command-Query Responsibility Segregation (CQRS) separates the actions that mutate state from the actions that query the state. Because mutations and queries typically exhibit contrasting runtime characteristics and require vastly different, often contradictory optimisation decisions, the separation of these concerns is conducive to building highly performant systems. The flip side is complexity — requiring multiple datastores and duplication of data — each datastore will maintain a unique projection of the master dataset, built from a dedicated data pipe. Kafka curbs some of the complexity inherent in CQRS architectures by acting as a common event-sourced ledger, using the concept of *consumer groups* to individually feed the different query-centric datastores.

This pattern is related to log shipping, and might appear identical at first glance. The differences are subtle. Log shipping is performed on low-level, internal representations of data, where both the replicas and the master datastore are coupled to, and share the same internal data structures. Put differently, log shipping is an internal mechanism that shuttles data within the confines of a single domain. In comparison, CQRS assumes disparate systems and spans domain boundaries. All parties are coupled to some canonical representation of events, which is versioned independently of the parties' internal representations of those events.

This chapter has introduced the reader to Apache Kafka — the world's most recognised and widely deployed event streaming platform. We looked at the history behind Kafka — how it started its journey and what it has become.

We also explored the various use cases that Kafka comfortably enables and supports. These are vast and varied, demonstrating Kafka's overall flexibility and eagerness to cater to a diverse range of event streaming scenarios.

Chapter 3: Architecture and Core Concepts

The first two chapters have furnished a cushy debut of the essential concepts of event streaming and have given the reader an introduction to Apache Kafka as the premier event streaming technology that is increasingly used to power organisations of all shapes and sizes — from green-sprout startups to multi-national juggernauts.

Now that the scene has been set, it is time to take a deeper look at how Kafka works, and more to the point, how one works with it.

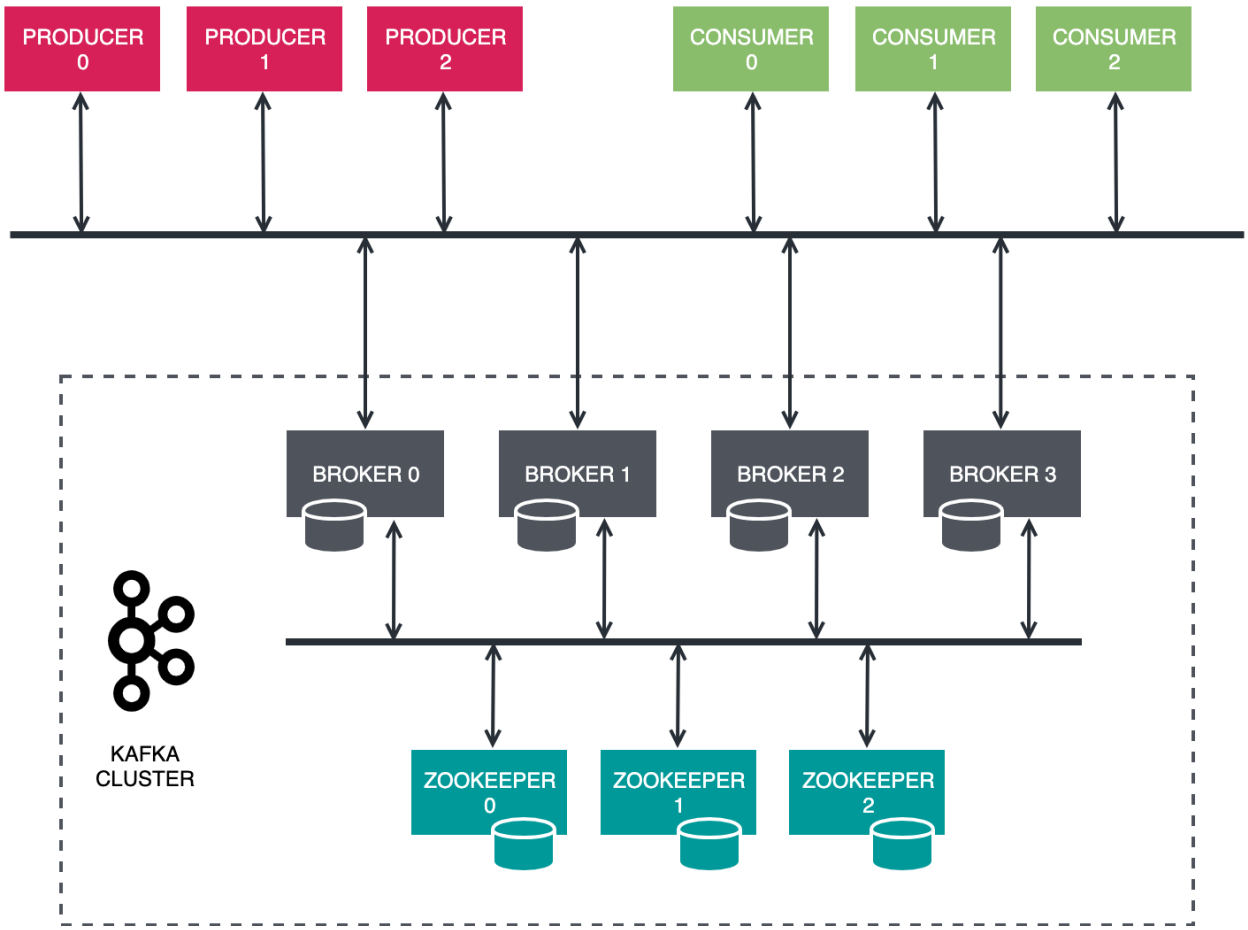
Architecture Overview

While the intention isn't to indoctrinate the reader with the minutia of Kafka's inner workings (for now), some appreciation of its design will go a long way in explaining the foundational concepts that will be covered shortly.

Kafka is a distributed system comprising several key components. At an outline level, these are:

- **Broker nodes**: Responsible for the bulk of I/O operations and durable persistence within the cluster.
- **ZooKeeper nodes**: Under the hood, Kafka needs a way of managing the overall controller status within the cluster. ZooKeeper fulfills this role, additionally acting as a consistent state repository that can be safely shared among the brokers.
- **Producers**: Client applications responsible for appending records to Kafka topics.
- **Consumers**: Client applications that read from topics.

The diagram below offers a brief overview of the Kafka component architecture, illustrating the relationships between its constituent parts. A further elaboration of the components follows.



Kafka Architecture Overview

Broker nodes

Before we begin, it is worth noting that industry literature uses the terminology 'Kafka Server', 'Kafka Broker' and 'Kafka Node' interchangeably to refer to the same concept. The official documentation refers to all three, while the shell scripts for starting Kafka and ZooKeeper refer to both as 'server'. This book favours the term 'broker' or, in some cases, the more elaborate 'broker node', avoiding the use of 'server' for its ambiguity.

So, what is a broker? If the reader comes from a background of messaging and middleware, the concept of a *broker* should resonate innately and intuitively. Otherwise, the reader is invited to consider Wikipedia's definition:

> *A broker is a person or firm who arranges transactions between a buyer and a seller for a commission when the deal is executed.*

Sans the commission piece, the definition fits Kafka like a glove. We would, of course, substitute 'buyer' and 'seller' for 'consumer' and 'producer', respectively, but one point is clear — the broker acts as an intermediary, facilitating the interactions between two parties — adding value in between.

This might make one wonder: *Why couldn't the parties interact directly?* A comprehensive answer would bore into the depths of computer science, specifically into the notion of *coupling*. We are not going to do this, to much relief; instead, the answer will be condensed to the following: the parties might not be aware of one another or they might not be jointly present at the same point in time. The latter places an additional demand on the broker: it must be stateful. In other words, it must persist the records emitted by the producer, so that they may be eventually delivered to the consumer when it is convenient to do so. The broker needs to be not only persistent, but also *durable*. By 'durable', it is implied that its persistence guarantees can be extended over a period of time and canvas scenarios that involve component failure.

Discussions of brokers as a means of decoupling communicating parties may conjure images of message queues from the days of yore. Many Kafka purists would protest: *Kafka is not a message queue, but an event streaming platform*. While the argument holds on the whole, the underpinning objectives remain largely unchanged. Fundamentally, we still have a publishing party, a subscribing party, and an intermediary to facilitate their interaction. And we would ideally like the parties to remain minimally coupled.

A Kafka broker is a Java process that acts as part of a larger cluster, where the minimum size of the cluster is one. (Indeed, we often use a singleton cluster for testing.) A broker is *one* of the units of scalability in Kafka; by increasing the number of brokers, one can achieve improved I/O, availability, and durability characteristics. (There are other ways of scaling Kafka, as we shall soon discover.)

A broker fulfills its persistence obligations by hosting a set of append-only log files that comprise the *partitions* hosted by the cluster. A more thorough discussion on partitions is yet to come; it will suffice to say for now that partitions are elemental units of storage that one can address in Kafka.

Each partition is mastered by exactly one broker — the partition *leader*. Partition data is replicated to a set of zero or more *follower* brokers. Collectively, the leader and the followers are referred to as *replicas*. Brokers share the load of leader and follower roles among themselves; a broker node may act as the leader for certain replicas, while being a follower for others. The roles may change — a follower replica may be promoted to leader status in the event of failure or as part of a manual rebalancing operation. The notion of replicas satisfies the durability guarantee; the more replicas in a cluster, the lower the likelihood of data loss due to an isolated replica failure.

Broker nodes are largely identical in every way; each node competes for the mastership of partition data on equal footing with its peers. Given the symmetric nature of the cluster, Kafka requires a mechanism for arbitrating the roles within the cluster and assigning partition leadership statuses among the broker nodes. Rather than making these decisions collectively, broker nodes follow a rudimentary chain-of-command. A single node is elected as the *cluster controller* which, in turn, directs all nodes (including itself) to assume specific roles. In other words, it is the controller's responsibility for managing the states of partitions and replicas, and for performing administrative tasks like reassigning partitions among the broker nodes.

ZooKeeper nodes

While the controller is entrusted with key administrative operations, the responsibility for electing a controller lies with another party — ZooKeeper. In fact, ZooKeeper is itself a cluster of cooperating processes called an *ensemble*. Every broker node will register its intent with ZooKeeper, but only one will be elected as the controller. ZooKeeper ensures that at most one broker node will be assigned the controller status, and should the controller node fail or leave the cluster, another broker node will promptly take its place.

A ZooKeeper ensemble also acts as a consistent and highly available configuration repository of sorts, maintaining cluster metadata, leader-follower states, quotas, user information, access control lists, and other housekeeping items. Owing to the underlying gossiping and consensus protocol of the ZooKeeper ensemble, *the number of ZooKeeper nodes must be odd.*

While ZooKeeper is bundled with Kafka for convenience, it is important to acknowledge that ZooKeeper is not an internal component of Kafka, but an open-source project in its own right.

Producers

A Kafka producer is a client application that can act as a source of data in a Kafka cluster. A producer communicates with the cluster over a set of persistent TCP connections, with an individual connection established with each broker. Producers can publish records to one or more Kafka topics, and any number of producers can append records to the same topic. Generally speaking, only producers are allowed to append records to topics; a consumer cannot modify a topic in any way.

Consumers

A consumer is a client application that acts as a data sink, subscribing to streams of records from one or more topics. Consumers are conceptually more complex than producers — they have to coordinate among themselves to balance the load of consuming records and track their progress through the stream.

Total and partial order

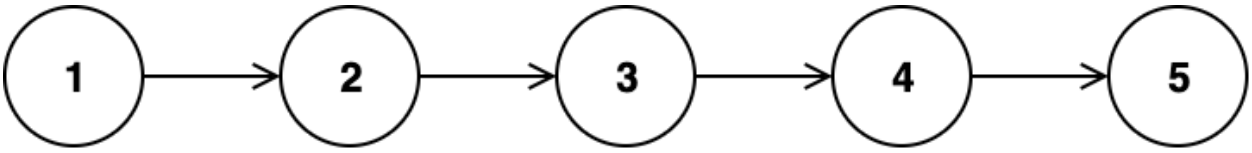
A minor note before we proceed: there is some theory ahead that one must endure to become an effective purveyor of Kafka. The upcoming content may seem esoteric and somewhat detached from the subject matter at first; however, the reader is assured that it is most relevant. We will be as brief as possible.

Without exaggeration, Kafka's entire event processing architecture is largely underpinned by the two primordial attributes of set theory: *partial order* and *total order*.

On the topic of set theory, what is a *set*? *A set is a collection of distinct elements* — objects that exist in their own right. For example, the numbers *2*, *4*, and *6* are distinct objects; when they are considered

collectively, they form a set of size three, written *{2, 4, 6}*. Developed at the end of the 19th century, set theory is now a ubiquitous part of mathematics; it is also generally considered fundamental to the construction of distributed and concurrent systems.

A *totally ordered* set is one where every element has a well-defined ordering relationship with every other element in the set. Consider, for example, the range of natural numbers one to five. When sorted in increasing order, it forms the following sequence:



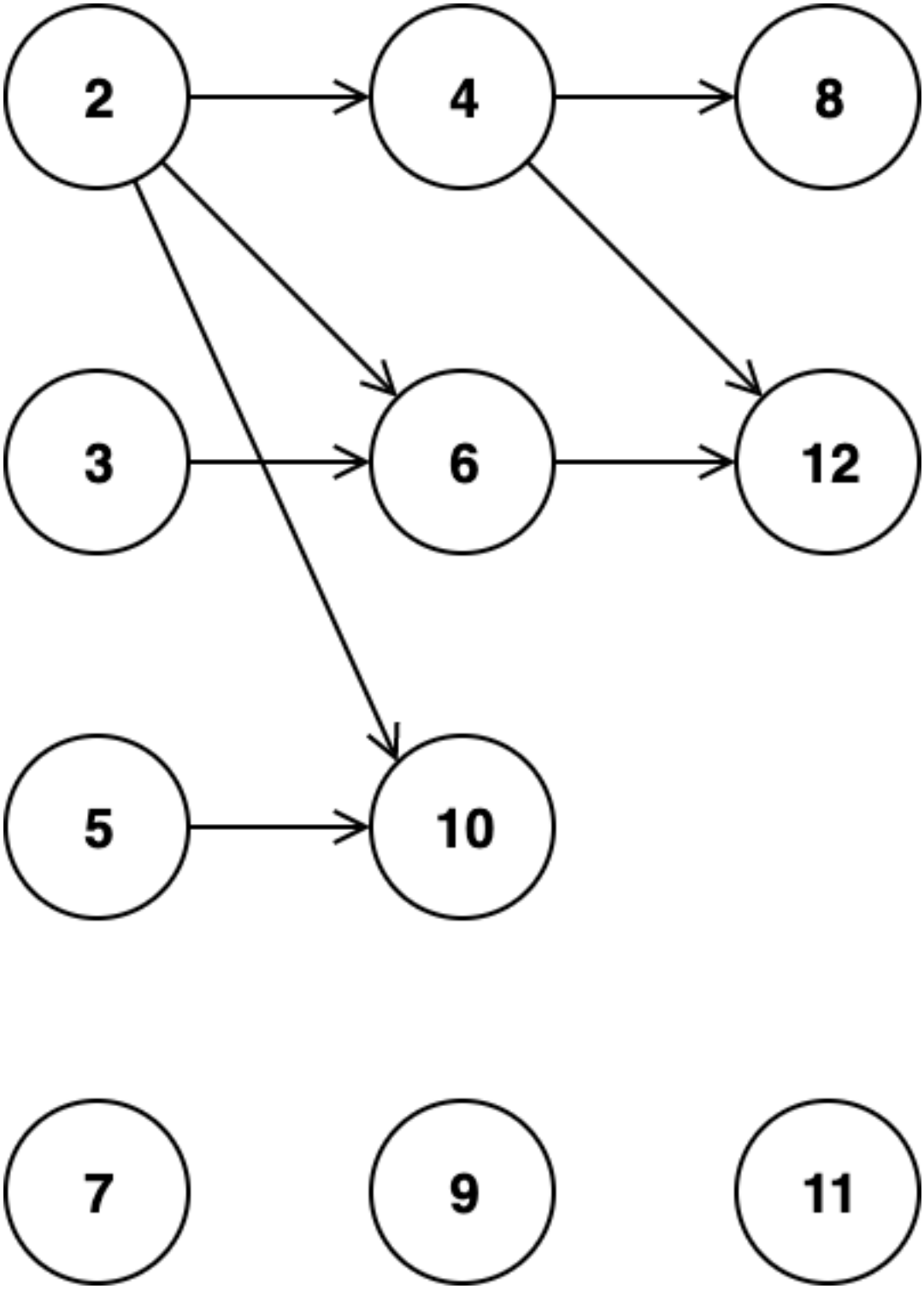
Totally ordered set

This is an ordered set, as every element has a well-defined predecessor-successor relationship with every other element. One can remove arbitrary values from this set and reinsert those values back into the set and arrive at the same sequence, no matter how many times this is attempted. Stated otherwise, there is only one permutation of elements that satisfies the ordering constraints.

Ordered sets exhibit the convenient property of *transitivity*. From the above example, we know that *2* must come after *1* and before *3*. We also know that *3* must come before *4*. Therefore, we can use the transitivity relation to deduce that *4* must come after *2*.

The direct antithesis of a totally ordered set is an unordered set. For example, an offhand list of capital cities *{Sydney, New York, London}* is unordered. Without applying further constraints, one cannot reason whether *Sydney* should appear before or after *New York*. One can arbitrarily permute the elements to arrive at different sequences of cities without upsetting anyone.

Between the two extremes, we find a partially ordered set. Consider the set of natural numbers ordered by divisibility, such that a number must appear after its divisor. For the range of numbers two to twelve, one instantiation of a sequence that satisfies this partial ordering constraint might be *[2, 3, 5, 7, 11, 4, 6, 9, 10, 8, 12]*. But that is just one instance — there are several such sequences that are distinct, yet equivalent. Looking at the set, we can state that the numbers *4* and *6* must appear after *2*, but there is no predecessor-successor relationship between *4* and *6* — they are mutually incomparable.



Partially ordered set

Multiple totally ordered sets can be contained in a single partially ordered set. For example, consider the Latin and Cyrillic alphabet sets *{A, B, C, …, Z}* and *{А, Б, В, …, Я}*, with their elements (letters)

arranged in alphabetical order — forming two distinct, totally ordered sets. Their union would be a partially ordered set; it would still maintain the relative order *within* each alphabet, without imposing order *across* alphabets.

Like in a totally ordered set, the elements of partially ordered sets exhibit transitivity. In the example involving divisors, the number *2*, appearing before *4*, where *4* appears before *8* or *12*, implies that both *8* and *12* must appear after *2*.

Topping off this discussion is the term 'causal order'. This type of ordering was first brought up in Chapter 1: Event Streaming Fundamentals, as part of the discussion on the consistency of replicated state. Unlike the other flavours, causal order is not a carryover from 19th-century mathematics; it stems from the study of distributed systems. A notable challenge of constructing such systems is that messages sent between processes may arrive zero or more times at any point after they are sent. As a consequence, there is no agreeable notion of time among collaborating processes. If one process, such as a clock, sends a timestamped message to another process, there is no reliable way for a receiver to determine the time relative to the clock and synchronise the two processes. This is often cited as the sole reason that building distributed systems is hard.

In the absence of a global clock, the relative timing of a pair of events occurring in close succession may be indistinguishable to an outside observer; however, if the two events are causally related, it is possible to distinguish their order; in other words, they become comparable. Causal order is a semantic rendition of partial order, where two elements may be bound by a *happened-before* relationship. This is denoted by an arrow appearing between the two elements; for example, if $A \longrightarrow B$, then A is an event that must logically precede B. This further implies that A occurred before B in a chronological sense. Otherwise, if $not(A \longrightarrow B)$, A cannot have preceded B in a causal sense. The latter does not imply that A could not have physically occurred before B; one simply has no way of ascertaining this.

> Causal relationships in distributed systems do not necessarily correspond to the more prevalent deductive 'cause-and-effect' style of logical reasoning. A causal relationship between a pair of events simply implies that one event precedes, rather than induces, the other. And it may be that the original events themselves are *not* comparable, but the recorded observations of these events are. These observations are events in their own right, and may also exhibit causality.
>
> Consider, for example, two samples of temperature readings *R0* and *R1* taken at different sites. They are communicated to a remote receiver and recorded in the order of arrival, forming a causal relationship on the receiver. If the message from *R0* was received first, we could confidently state that $received(R0) \longrightarrow received(R1)$. This does not imply that $sent(R0) \longrightarrow sent(R1)$, and it most certainly does not imply that *R0* played any part in inducing *R1*.

In considering the connection between the terms 'ordered', 'unordered', 'partially ordered', 'causally ordered', and 'totally ordered', one can draw the following synopsis:

- A partially ordered set implies that not every pair of elements needs to be comparable.

- A totally ordered set is a special case of a partially ordered set, where there exists a well-defined order between every conceivable element pair.
- An unordered set is also a special case of a partially ordered set, where there is no pair of comparable elements.
- Causal order is a rendition of partial order, where each element represents an event, and some pairs of events have a happened-before relationship.
- On its own, the term 'ordered set' is ambiguous, suggesting that the elements of a set exhibit some order-inducing relationships. This term is generally avoided.

With partial and total order out of the way, we can proceed to a discussion of records, topics, and partitions. The link between the latter and set theory will shortly become apparent.

Records

A *record* is the most elemental unit of persistence in Kafka. In the context of event-driven architecture, which is chiefly how one is meant to use Kafka, a record typically corresponds to some event of interest. It is characterised by the following attributes:

- **Key**: A record can be associated with an optional non-unique key, which acts as a kind of classifier — grouping related records on the basis of their key. The key is entirely free-form; anything that can be represented as an array of bytes can serve as a record key.
- **Value**: A value is effectively the informational payload of a record. The value is the most interesting part of a record in a business sense — it is the record's value that ultimately describes the event. A value is optional, although it is rare to see a record with a `null` value. Without a value, a record is largely pointless; all other attributes play a supporting role in conveying the value.
- **Headers**: A set of free-form key-value pairs that can optionally annotate a record. Headers in Kafka are akin to their namesake in HTTP — they augment the main payload with additional metadata.
- **Partition number**: A zero-based index of the partition that the record appears in. A record must always be tied to exactly one partition; however, the partition need not be specified explicitly when the record is published.
- **Offset**: A 64-bit signed integer for locating a record within its encompassing partition. Records are stored sequentially; the offset represents a logical sequence number of the record.
- **Timestamp**: A millisecond-precise timestamp of the record. A timestamp may be set explicitly by the producer to an arbitrary value, or it may be automatically assigned by the broker when a record is appended to the log.

Newcomers to Kafka usually have no problems grasping the concept of a record and understanding its internals, with the possible exception of the *key* attribute. Because Kafka is often likened to a database (albeit one for storing events), a record's key is often incorrectly associated with a database

key. This warrants prompt clarification, so as to not cause confusion down the track. Kafka does have a primary key, but it is *not* the record key. A record's equivalent of a 'primary key' is the composition of the record's partition number and its offset. A record's *key* is not unique, and therefore cannot possibly serve as the primary key. Furthermore, Kafka does not have the concept of a secondary index, and so the record key cannot be used to isolate a set of matching records.

Instead, it is best to think of a key as a kind of a pigeonhole into which related records are placed. Records maintain an association with their key over their entire lifetime. One cannot alter the key, or any aspect of the record for that matter, once the record has been published. It is unfortunate that keys are named as they are; a *classifier* (or a synonym thereof) would have been more appropriate.

The reason why the term 'key' was chosen is likely due to its association with hashing. Kafka producers use keys to map records to partitions — an action that involves hashing of the key bytes and applying the modulo operator. This carries a close resemblance to how keys are hashed to yield a bucket in a hash table.

The correspondence between Kafka records and observed events may not be direct; for example, an event might spawn multiple Kafka records, typically emitted in close succession. Those records may, in turn, be processed by a staged event-driven pipeline — spawning additional records in the course of processing. An overview of the staged event-driven architecture (SEDA) pattern was presented in Chapter 2: Introducing Apache Kafka.

Kafka is often used as a communication medium between fine-grained application services or across entire application domains. In saying that, the employment of Kafka as an internal note-taking or ledgering mechanism within a bounded context, is a perfectly valid use case. In fact, both *Event Sourcing* and *CQRS* patterns have seen strong adoption within the confines of single a domain, as well as across domains.

The recorded event might not have a real-life equivalent, even indirectly or circumstantially; there is no assumption or implication that Kafka is used solely as a registry of events. This statement may ruffle a few feathers or spark an all-out *bellum sacrum*; after all, Kafka is an event streaming platform — if not for recording events, what could it possibly be used for? Well, it might be used to replace a more traditional message broker. Much to the despise of Kafka purists (or delight, depending on one's personal convictions), an increasingly-growing use for Kafka is to replace technologies such as RabbitMQ, ActiveMQ, AWS SQS and SNS, Google Cloud Pub/Sub, and so forth. Kafka renowned flexibility lets it comfortably deal with a broad range of messaging topologies and applications, some of which have little resemblance to classical event-driven architecture.

> The generally accepted relationship between the terms 'message' and 'event' is such that a message encompasses a general class of asynchronous communiqués, while an event is a semantic specialisation of a message that communicates that some action of significance has occurred. Events have one logical owner — the producer (or publisher); they are immutable; they can be subscribed to and unsubscribed from. The term 'event' is often contrasted with another term — 'command' — a specialised message encompassing a directive issued from one party to another, requesting it to

perform some action. The logical owner of a command is its sole recipient; it cannot be subscribed to or unsubscribed from.

Kafka documentation and client APIs mostly prefer the term 'record', where others might use 'message' or 'event'. Kafka literature occasionally uses 'message' as a substitute for 'record', but this has been generally discouraged within the community, for the angst of confusing Kafka with the more traditional message-oriented middleware. In this book, the term 'record' is preferred, particularly when working in the context of event streaming. The term 'event' will generally be used to refer to an external action that triggered the publishing of the record, but may also be metonymically used to refer to the record itself, as it is often convenient to do so. Finally, this book may occasionally use the term 'message' when describing records in the context of a more traditional message broker, where the use of this term aids clarity.

Partitions

A partition is a totally ordered, unbounded set of records. Published records are appended to the head-end of the encompassing partition. Where a record can be seen as an elemental unit of persistence, a partition is an elemental unit of record streaming.

Because records are totally ordered within their partition, any pair of records in the same partition is bound by a predecessor-successor relationship. This relationship is implicitly assigned by the producer application. For any given producer instance, records will be written in the order they were emitted by the application. By way of example, assume a pair of records *P* and *Q* destined for the same partition. If record *P* was published before *Q*, then *P* will precede *Q* in the partition. Furthermore, they will be read in the same order by *all* consumers; *P* will always be read before *Q*, for every possible consumer. This ordering guarantee is vital when implementing event-driven systems, more so than for peer-to-peer messaging or work queues; published records will generally correspond to or derive from real-life events, and preserving the timeline of these events is often essential.

Records published to one partition by the same producer are causally ordered. In other words, if *P* precedes *Q*, then *P* must have been observed before *Q* on the producer; the happened-before relationship is preserved — imparted from the producer onto the partition.

There is no recognised causal ordering *across* producers; if two (or more) producers emit records simultaneously for the same partition, those records may materialise in arbitrary order. The relative order will not depend on which producer application attempted to publish first, but rather, which record beat the other to the partition leader. That said, whatever the order, it will be *consistent* — observed uniformly across all consumers. Total order is still preserved, but some record pairs may only be related circumstantially.

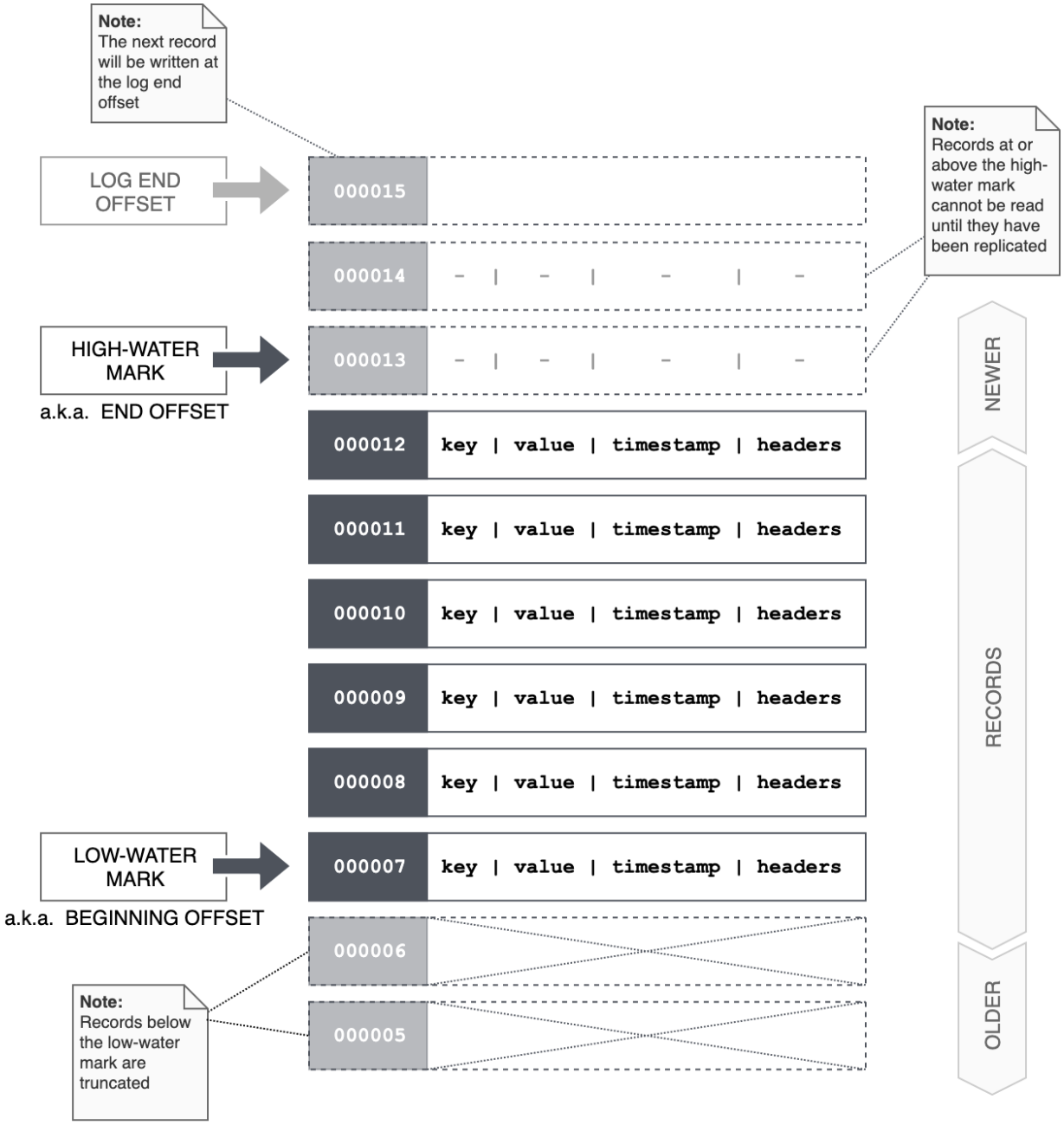
Corollary to the above, *in the absence of producer synchronisation, causal order can only be achieved when a single producer emits records to the same partition.* All other combinations — involving

multiple unrelated producers or different partitions — may result in a record stream that fails to depict causality. Whether or not this is an issue will depend largely on the application.

A record's offset uniquely identifies it in the partition. The offset acts as a primary key, allowing for fast, *O(1)* lookups. The offset is a strictly monotonically-increasing integer in a sparse address space, meaning that each successive offset is always higher than its predecessor, and there may be varying gaps between neighbouring offsets. Gaps might legitimately appear if compaction is enabled or as a result of transactions; we don't need to delve into the details at this stage, suffice it to say that offsets need not be contiguous.

Given a record, an application shouldn't attempt to literally interpret its offset or guess what the next offset might be. It may, however, actively exploit the properties of *total order* and *transitivity* to infer the relative order of any record pair based on their offsets, sort the records by their offset, and so forth.

The diagram below shows what a partition looks like on the inside.



Partition contents

The *beginning offset*, also called the *low-water mark*, is the first record that will be presented to a prospective consumer. Due to Kafka's bounded retention, this is not necessarily the first record that was published. Records may be pruned on the basis of time and/or partition size. When this occurs, the low-water mark will appear to advance, and records earlier than the low-water mark will be truncated.

Conversely, the *high-water mark* is the offset immediately following the last successfully replicated record. Consumers are only allowed to read up to the high-water mark. This prevents a consumer

from reading unreplicated data that may be lost in the event of leader failure. The equivalent term for a high-water mark is the *end offset*.

The statement above is a minor simplification. The end offset corresponds to the high-water mark for non-transactional consumers. Where a more strict isolation mode has been selected on the consumer, the end offset may trail the high-water mark. Transactional messaging is an advanced topic, covered in Chapter 18: Transactions.

The end offset should not be confused with the internal term 'log end offset', which is the offset immediately following that of the last written record. The 'log end offset' will be assigned to the next record that will be published. When the follower replicas lag behind the leader, the 'log end offset' will be greater than the high-water mark. When replication eventually catches up, the high-water mark will align with the 'log end offset'.

Subtracting the low-water mark from the high-water mark will yield the upper bound on the number of securely persisted records in the partition. The actual number may be slightly less, as the offsets are not guaranteed to be contiguous. The *total* number of records may be fewer or greater, as the high-water mark does not reflect the number of unreplicated records.

Topics

So, a partition is an unbounded sequence of records, an open ledger, a continuum of events — each definition as good as the next. Along with a record, a partition is an elemental building block of an event streaming platform. But a partition is too basic to be used effectively on its own.

A *topic* is a logical aggregation of partitions. It comprises one or more partitions, and a partition must be a part of exactly one topic. Topics are fundamental to Kafka, allowing for both parallelism and load balancing.

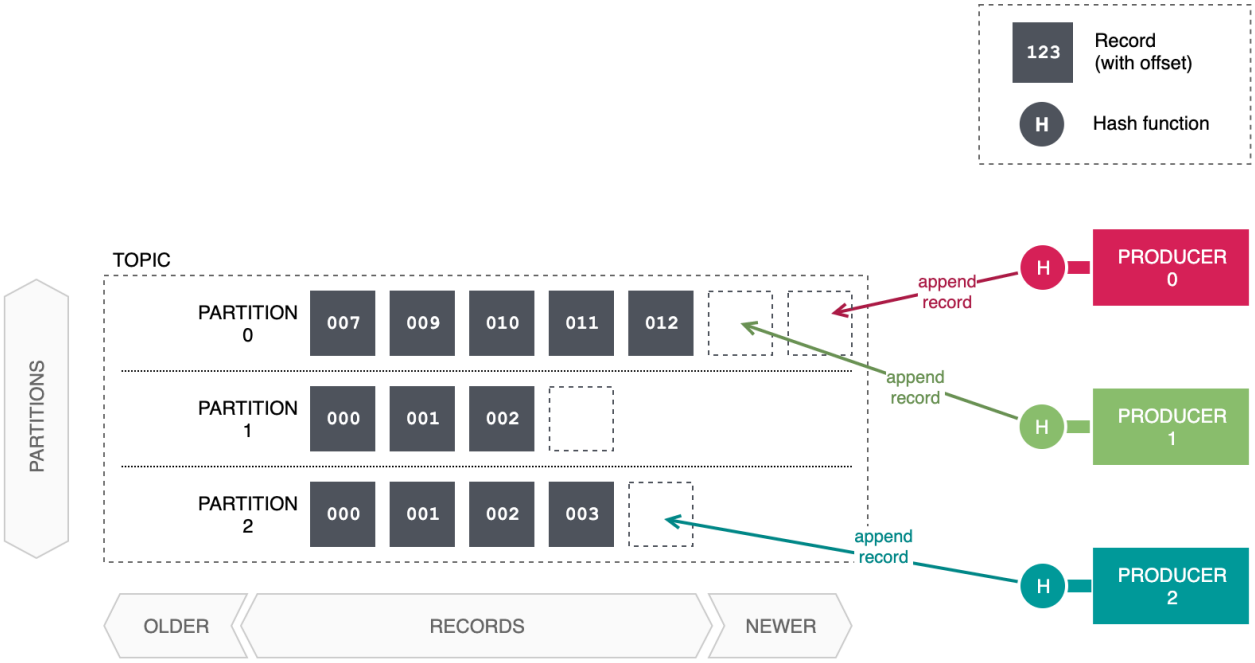
Earlier, it was said that partitions exhibit total order. Taking a set-theoretic perspective, a topic is just a union of the individual underlying sets; since partitions within a topic are mutually independent, the topic is said to exhibit *partial order*. In simple terms, this means that certain records may be ordered in relation to one another, while being unordered with respect to certain other records. A Kafka topic, and specifically its use of partial order, enables us to process records in parallel *where we can*, while maintaining order *where we must*. The concept of consumer parallelism will be explored shortly; for the time being, the focus will remain on the producer ecosystem.

Since Kafka is an event streaming platform, it may be more instructive to think of a topic and its partitions as a wide *stream*, comprising multiple parallel *substreams*. Events within a substream *may* be related objectively, insofar as one event must precede some other event. In other words, a causal relationship is in place. (The events are not required to be causally related to share a substream; the reason that will be touched on later.) Events across substreams are related subjectively — they might

refer to a similar class of observations and it may be advantageous to encompass them within the same stream.

Occasionally, this book will use the term 'stream' as a substitute for 'topic'; when referring to events, the use of the term 'stream' is often more natural and intuitive.

Precisely how records are partitioned is left to the discretion of the producer. A producer application may explicitly assign a partition number when publishing a record, although this approach is rarely used. A much more common approach is for the client application to deal exclusively with record keys and values, and have the producer library automatically select a partition on the basis of a record's key. A producer will digest the byte content of the key using a hash function (Kafka uses `murmur2` for this purpose). The highest-order bit of the hash value is masked off to force it to a positive integer, before taking the result, modulo the number of partitions, to arrive at the final partition number. The contents of the topic and the producers' interactions with the topic are depicted below.



Topic contents

While this partitioning scheme is deterministic, it is not *consistent*. Two records with the same key hashed at different points in time will correspond to an identical partition number *if and only if* the number of partitions has not changed in that time. Increasing the number of partitions in a topic (Kafka does not support non-destructive downsizing) results in the two records occupying potentially different partitions — leading to a breakdown of any prior order. There are many gotchas, such as this one, in Kafka; they will be called out as such from time to time.

Records sharing the same hash are guaranteed to occupy the same partition. Assuming a topic with

multiple partitions, records with a different key will likely end up in different partitions. However, due to hash collisions, records with different hashes may also end up in the same partition. Such is the nature of hashing; if the reader appreciates how a hash table works, this is no different. It was previously stated that records in the same partition may be causally related, but do not have to be. The reason is specifically to do with hashing; when there are more causally related record groupings then there are partitions in a topic, there will invariably be some partitions that contain multiple unrelated sets of records. In mathematics, this is referred to as the *Dirichlet's Drawer Principle* or the *Pigeonhole Principle*. In fact, due to the imperfect space distribution of hash functions, unrelated records will likely be grouped in the same partition even there are more partitions than distinct keys.

Producers rarely care which specific partition the records will map to, only that related records end up in the same partition, and that their order is preserved. Similarly, consumers are largely indifferent to their assigned partitions, so long that they receive the records in the same order as they were published, where those records are causally bound.

Consumer groups and load balancing

So far we have learned that producers emit records to a topic; these records are organised into neatly ordered partitions. Kafka's producer-topic-consumer topology adheres to a flexible and highly generalised *multipoint-to-multipoint* model, meaning that there may be any number of producers and consumers simultaneously interacting with a topic. Depending on the actual solution context, topologies may also be point-to-multipoint, multipoint-to-point, and point-to-point. Kafka does not impose the sorts of limits that one is used to seeing from the more 'orthodox' messaging middleware. It's about time we looked at how records are consumed.

A *consumer* is a process or thread that attaches to a Kafka cluster via a client library. A consumer generally, but not necessarily, operates as part of an encompassing *consumer group*. Consumer groups are effectively a *load-balancing* mechanism within Kafka — distributing partition assignments approximately evenly among the individual consumer instances within the group. When the first consumer in a group subscribes to the topic, it will receive all partitions in that topic. When a second consumer subsequently joins, it will get approximately half of the partitions, relieving the first consumer of half of its prior load. The process runs in reverse when consumers leave (by disconnecting or timing out) — the remaining consumers will absorb a greater number of partitions.

This book will occasionally use the term 'subscriber' to collectively refer to all consumer instances in a consumer group, as a single, logical entity. When referring to event streams, the notion of a subscriber is sometimes more intuitive and helps distinguish between those consumers who may not be a part of a consumer group at all. Those sorts of consumers will be discussed later.

So, a consumer siphons records from a topic, pulling from the share of partitions that have been assigned to it by Kafka, alongside the other consumers in its group. As far as load-balancing goes, this is nothing out of the ordinary. But here's the kicker — *consuming a record does not remove it from*

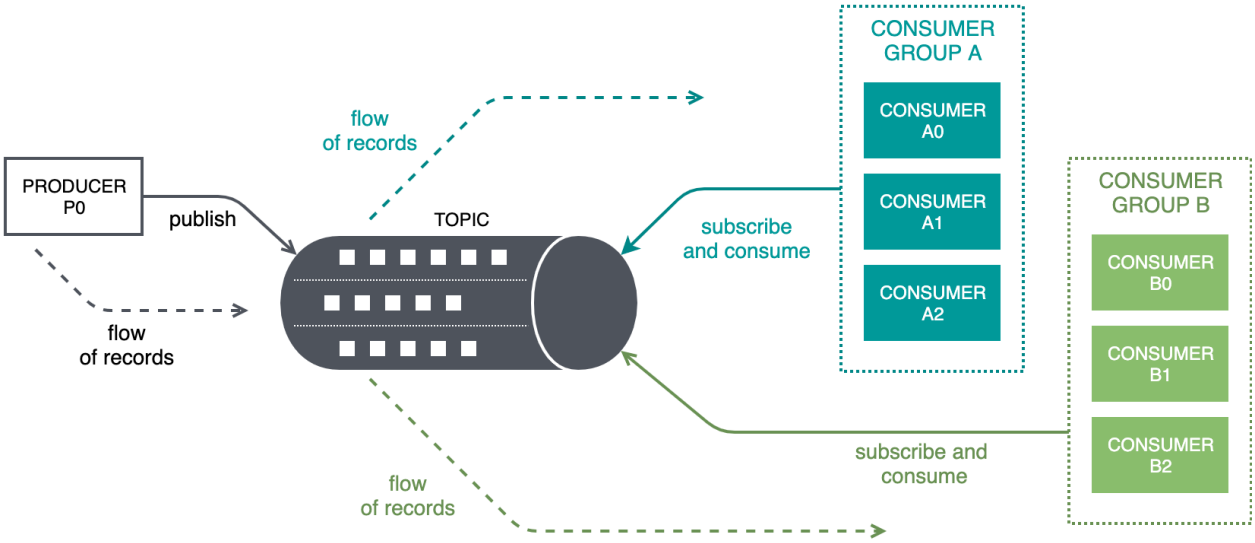
the topic. This might seem contradictory at first, especially if one associates the act of consuming with depletion. (If anything, a consumer should have been called a 'reader', but let's not dwell on the choice of terminology.) The simple fact is, consumers have absolutely no impact on the topic and its partitions; a topic is an append-only log that may only be mutated by the producer, or by Kafka itself as part of its housekeeping chores. Consumers are 'cheap', so to speak — you can have a fair number of them tail the logs without stressing the cluster. This is a yet another point of distinction between an event stream and a traditional message queue, and it's a crucial one.

A consumer internally maintains an offset that points to the next record in a partition, advancing the offset for every successive read. In fact, a consumer maintains a vector of such offsets — one for each assigned partition. When a consumer first subscribes to a topic, whereby no offsets have been registered for the encompassing consumer group, it may elect to start at either the head-end or the tail-end of the topic. Thereafter, the consumer will acquire an offset vector and will advance the offsets internally, in line with the consumption of records.

In Kafka terminology, the 'head' of a partition corresponds to the location of the end offsets, while the 'tail' of the partition is the side closest to the beginning offsets. This might sound confusing if Kafka is perceived as a queue of sorts, where the head-end of a queue canonically corresponds to the side which has the oldest elements. In Kafka, the oldest elements are at the tail-end.

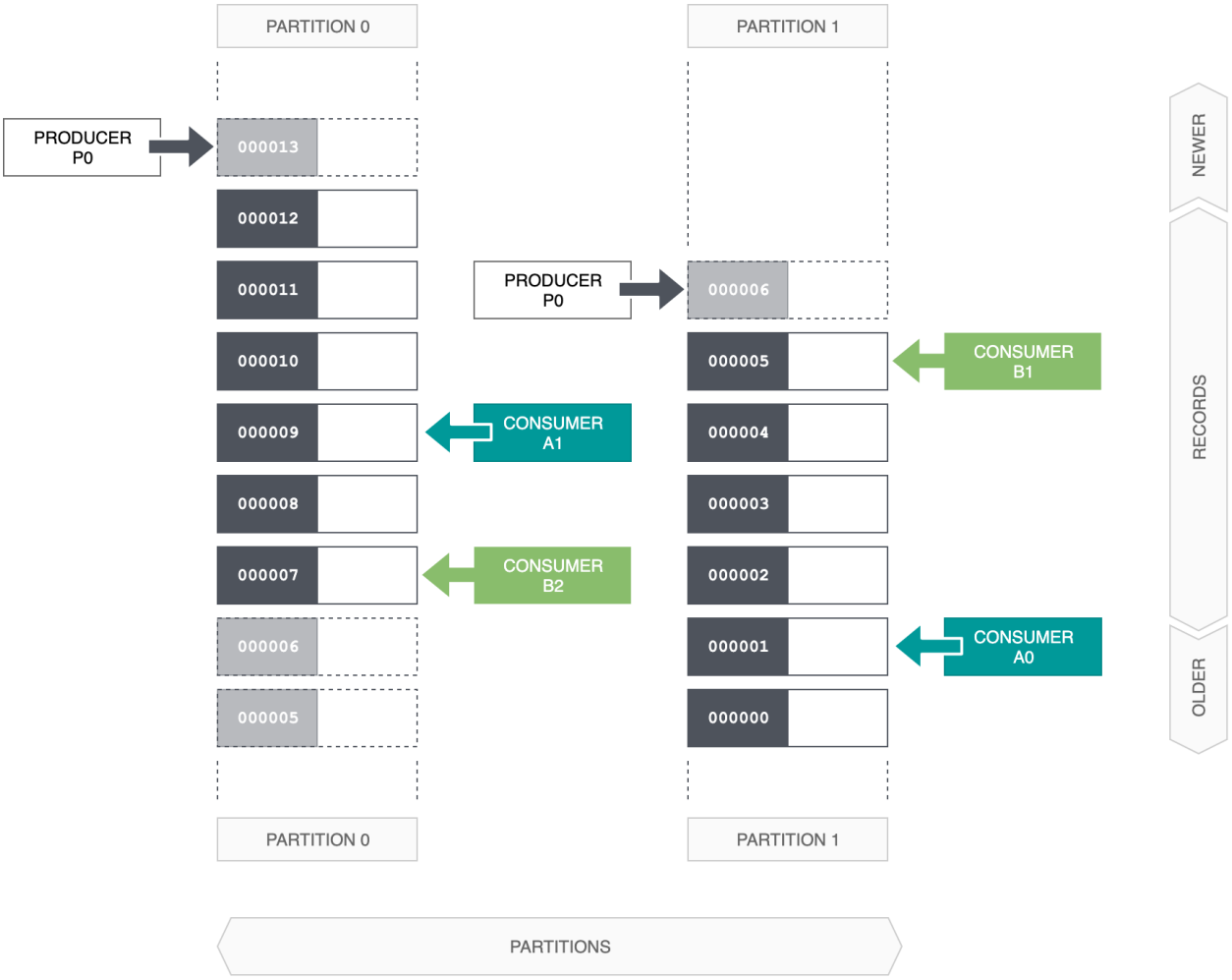
Since consumers across different consumer groups do not interfere, there may be any number of them reading concurrently from the same topic. Consumers run at their own pace; a slow or backlogged consumer has no impact on its peers.

To illustrate this concept, consider a contrived scenario involving a topic with two partitions. Two consumer groups — *A* and *B* — are subscribed to the topic. Each group has three consumer instances, named *A0*, *A1*, *A2*, *B0*, *B1*, and *B2*. The relationship between topics, consumers, and groups is illustrated below.



Multiple consumer groups sharing a topic

As part of fulfilling the subscriptions, Kafka will allocate partitions among the members of each group. In turn, each group will acquire and maintain a dedicated set of offsets that reflect the overall progress of the group through the topic. The sharing of the topic and the independent progress of consumer groups is diagrammatically depicted below.



Consumer offsets

Upon careful inspection, the reader will notice that something is missing. Two things, in fact: consumers *A2* and *B0* aren't there. That is because Kafka ensures that a partition may only be assigned to at most one consumer within its consumer group. (It is said 'at most' to cover the case when all consumers are offline.) Because there are three consumers in each group, but only two partitions, one consumer will remain idle — waiting for another consumer in its respective group to depart before being assigned a partition. In this manner, consumer groups are not only a load-balancing mechanism, but also a fence-like exclusion control, used to build highly performant pipelines without sacrificing *safety*, particularly when there is a requirement that a record may only be handled by one thread or process at any given time.

Consumer groups also ensure *availability*, satisfying the *liveness* property of a distributed consumer ecosystem. By periodically reading records from a topic, the consumer implicitly signals to the cluster that it is in a 'healthy' state, thereby extending the lease over its partition assignment. Should the consumer fail to read again within the allowable deadline, it will be deemed faulty and its partitions will be reassigned — apportioned among the remaining 'healthy' consumers within its group.

A thorough discussion of the *safety* and *liveness* properties of Kafka will be deferred until Chapter 15: Group Membership and Partition Assignment. For the time being, the reader is asked to accept an abridged definition: Liveness is a property that requires a system to eventually make progress, completing all assigned work. Safety is a property that requires the system to respect all its key invariants, at all times.

To employ a transportation analogy, a topic is like a highway, while a partition is a lane. A record is the equivalent of a car, and its occupants correspond to the record's value. Several cars can safely travel on the same highway, providing they keep to their lane. Cars sharing the same line ride in a sequence, forming an orderly queue. Now suppose each lane leads to an off-ramp, diverting its traffic to some location. If one off-ramp gets banked up, other off-ramps may still flow smoothly.

It is precisely this highway-lane metaphor that Kafka exploits to achieve its trademark end-to-end throughput, easily reaching millions of records per second on commodity hardware. When creating a topic, one can set the partition count — the number of lanes, if you will. The partitions are divided approximately evenly among the individual consumers in a consumer group, with a guarantee that no partition will be assigned to two (or more) consumers at the same time, providing that these consumers are part of the *same consumer group*. Referring to our analogy, a car will never end up in two off-ramps simultaneously; however, two lanes might conceivably merge to the same off-ramp.

The High Five Interchange, Dallas, Texas

Committing offsets

It has already been said that consumers maintain an internal state with respect to their partition offsets. At some point, that state must be shared with Kafka, so that when a partition is reassigned, the new consumer can resume processing from where the outgoing consumer left off. Similarly, if the consumers were to disconnect, upon reconnection they would ideally skip over any records that have already been processed.

Persisting the consumer state back to the Kafka cluster is called *committing* an offset. Typically, a consumer will read a record (or a batch of records) and commit the offset of the last record *plus one*. If a new consumer takes over the topic, it will commence processing from the last committed offset — hence the plus-one step is essential. (Otherwise, the last processed record would be handled a second time.)

Curious fact: Kafka employs a recursive approach to managing committed offsets, elegantly utilising itself to persist and track offsets. When an offset is committed, the group coordinator will publish a binary record on the internal `__consumer_offsets` topic. The contents of this topic are compacted in the background, creating an efficient event store that progressively reduces to only the last known commit points for any given consumer group.

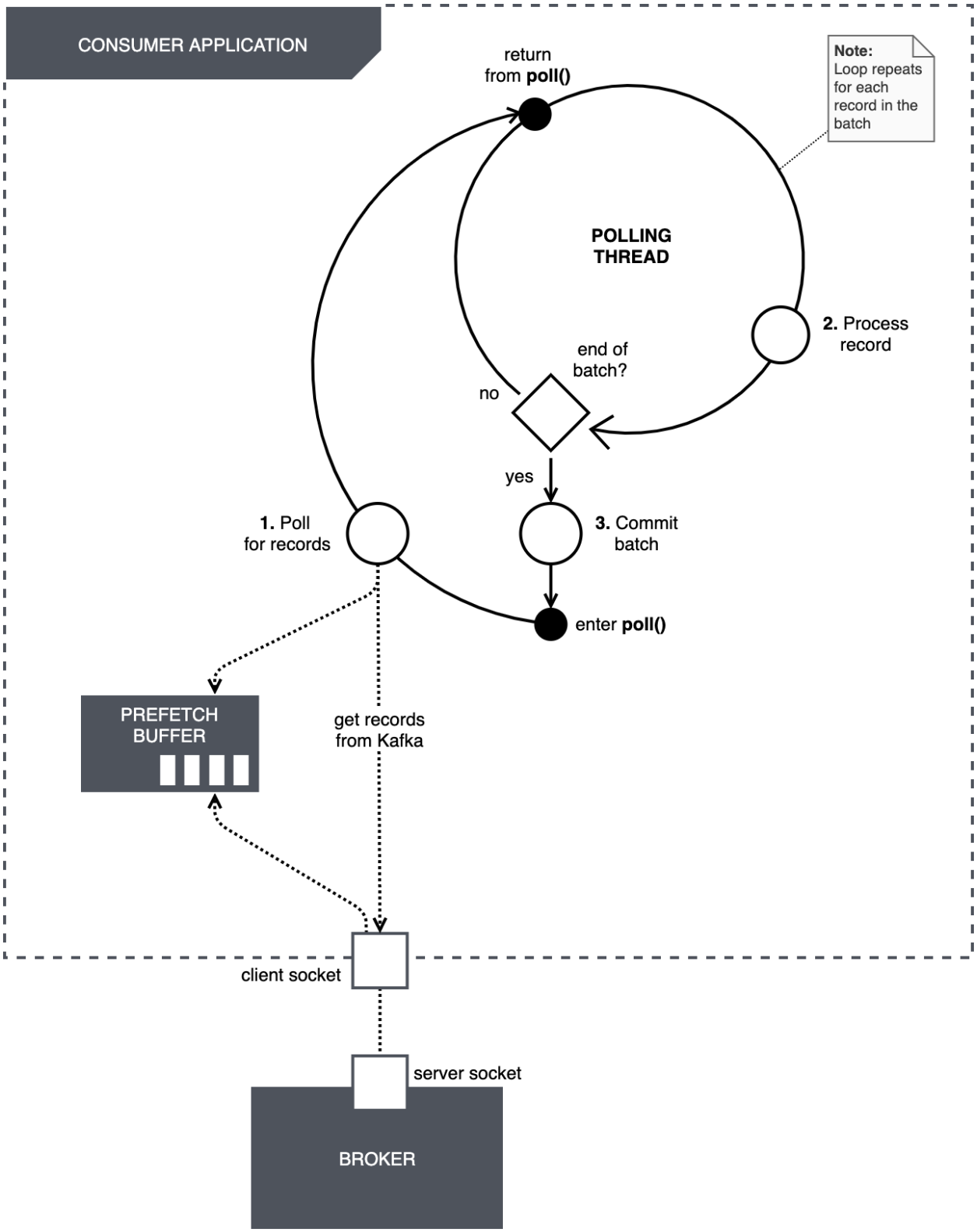
Controlling the point when an offset is committed provides a great deal of flexibility around *delivery guarantees*, further highlighting Kafka's adaptability towards various messaging scenarios. The term 'delivery' assumes not just reading a record, but the full processing cycle, complete with any side-effects. (For example, updating a database, or invoking a service.) One can shift from an *at-most-once* to an *at-least-once* delivery model by simply moving the commit operation from a point *before* the processing of a record is commenced, to a point sometime *after* the processing is complete. With this model, should the consumer fail midway through processing a record, it will be re-read following partition reassignment.

By default, a Kafka consumer will automatically commit offsets at an interval of *at least* every five seconds. The interval will automatically be extended in the presence of in-flight records — the records that are still being processed on the consumer. The lower bound on this interval can be controlled by the `auto.commit.interval.ms` configuration property, which is discussed in Chapter 10: Client Configuration. An implication of the offset auto-commit feature is that it extends the window of uncommitted offsets beyond the set of in-flight records; in other words, the consumer might finish processing a batch of records without necessarily committing the offsets. If the consumer's partitions are then reassigned, the new consumer will end up processing the same batch a second time. To constrain the window of uncommitted records, one needs to take offset committing into their own hands. This can be done by setting the `enable.auto.commit` client property to `false`.

Getting offset commits right can be tricky, and routinely catches out beginners. A committed offset implies that the record *one below that offset and all prior records have been dealt with by the consumer*. When designing at-least-once applications, an offset should only be committed when the application has dealt with the record in question, and all records before it. In other words, the record

has been processed to the point that any actions that would have resulted from the record have been carried out and finalised. This may include calling other APIs, updating a database, committing transactions, persisting the record's payload, or publishing more records. Stated otherwise, if the consumer were to fail after committing the record, then not ever seeing this record again must not be detrimental to its correctness.

In the at-least-once scenario, a typical consumer implementation will commit its offsets linearly, in tandem with the processing of a record batch. That is, read a record batch from a topic, process the individual records, commit the outstanding offsets, read the next batch, and so on. This is called a *poll-process* loop, illustrated below:



Consumer poll-process loop

The poll-process loop in the above diagram is a somewhat simplified take on reality. We will not go into the details of how records are fetched from Kafka when `KafkaConsumer.poll()` is called; a more thorough description is presented in Chapter 7: Serialization. We will remark on one optimisation: a consumer does not always fetch records directly from the cluster; it employs a prefetch buffer to pipeline this process.

A common tactic is to process a batch of records concurrently (where this makes sense), using a thread pool, and only confirm the last record when the entire batch is done. The commit process in Kafka is very efficient, the client library will send commit requests asynchronously to the cluster using an in-memory queue, without blocking the consumer. The client application can register an optional callback, notifying it when the commit has been acknowledged by the cluster. And there is also a blocking variant available should the client application prefer it.

Free consumers

The association of a consumer with a consumer group is an optional one, indicated by the presence of a `group.id` consumer property. If unset, a *free consumer* is presumed. Free consumers do not subscribe to a topic; instead, the consuming application is responsible for manually assigning a set of topic-partitions to itself, individually specifying the starting offset for each topic-partition pair. *Free consumers do not commit their offsets to Kafka*; it is up to the application to track the progress of such consumers and persist their state as appropriate, using a datastore of their choosing. The concepts of automatic partition assignment, rebalancing, offset persistence, partition exclusivity, consumer heartbeating and failure detection (safety and liveness, in other words), and other so-called 'niceties' accorded to consumer groups cease to exist in this mode.

The use of the nominal expression *'free consumer'* to denote a consumer without an encompassing group is a coined term. It is not part of the standard Kafka nomenclature; indeed, there is no widespread terminology that marks this form of consumer.

Free consumers are not observed in the wild as often as their grouped counterparts. There are predominantly two use cases where a free consumer is an appropriate choice. One such case is when an application genuinely requires full control of the partition assignment scheme, likely utilising a dedicated datastore to track consumer offsets. This is *very* rare. Needless to say, it is also difficult to implement correctly, given the multitude of scenarios one must account for. It is mentioned here only for completeness.

The more commonly seen use case is when a stateless or ephemeral consumer needs to monitor a topic. For example, an application might tail a topic to identify specific records, or just as a monitoring or debugging aid. One might only care about records that were published when the stateless consumer was online, so concerns such as persisting offsets and resuming from the last processed record become largely irrelevant. A good example of where this is used routinely is the Kafdrop tool, which we will explore in one of the upcoming chapters. When the user clicks on a

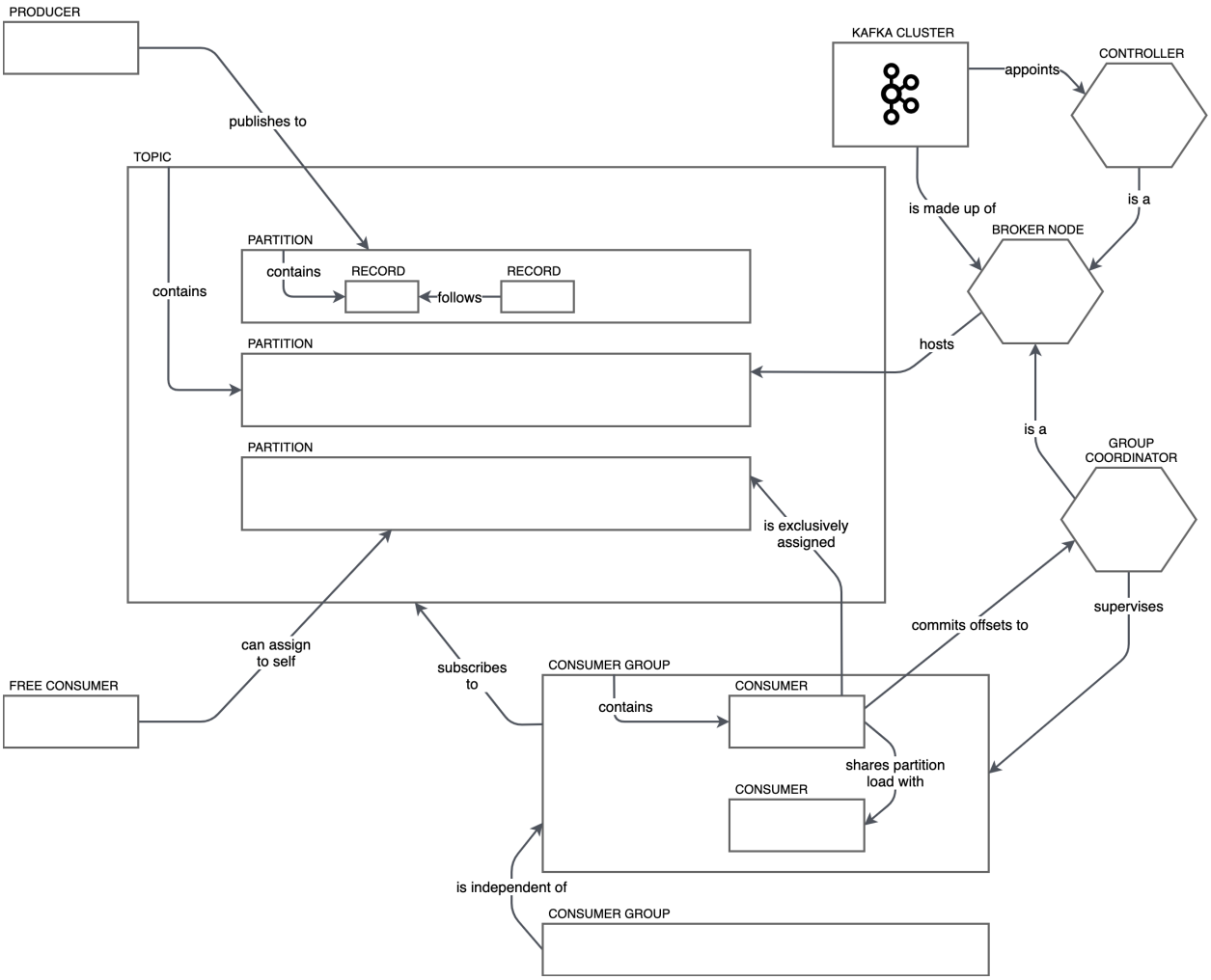
topic to view the records, Kafdrop creates a free consumer and assigns the requested partition to it, reading the records from the supplied offsets. Navigating to a different topic or partition will reset the consumer, discarding any prior state.

One scenario that benefits from free consumers is the implementation of the *sync-over-async* pattern using Kafka. For example, a producer might issue a command-style request (or query) to a downstream consumer and expect a response on the same or different topic. The initiating producer might be operating in a synchronous context; for example, it might be responding to a synchronous request of its own, and so it has no choice but to wait for the downstream response before proceeding. To complicate matters, there might be multiple such initiators in operation, and it is essential that the response is processed by the same initiator that issued the original request.

The sync-over-async scenario is a special case of the stateless consumer scenario presented above. The initiator starts by assigning itself all partitions of the response topic and resetting the offsets to the high-water mark. It then publishes the request command with a unique identifier that will be echoed in the response. (Typically, this is a UUID.) The downstream consumer will eventually process the message and publish its response. Meanwhile, the initiator will poll the topic for responses, filtering by ID. Eventually, either the response will arrive within a set deadline, or the initiator will time out. Either way, the assignment of the partitions to the initiator is temporary, and no state is preserved between successive assignments.

Summary of core concepts

The illustration below outlines the relationship between the core concepts presented in this chapter, including entities such as the cluster, broker nodes, producers, topics, partitions, consumers, and consumer groups.



Relationships between core concepts

The key takeaways are:

- **A cluster hosts multiple topics**, each having an assigned leader and zero or more follower replicas.
- **Topics are subdivided into partitions**, with each partition forming an independent, totally-ordered sequence within a wider, partially-ordered stream.
- **Multiple producers are able to publish to a topic**, picking a partition at will. The partition may be selected directly — by specifying a partition number, or indirectly — by way of a record key, which deterministically hashes to a partition number.
- **Partitions in a topic can be load-balanced** across a population of consumers in a consumer group, allocating partitions approximately evenly among the members of that group.
- **A consumer in a group is not guaranteed a partition assignment**. Where the group's population outnumbers the partitions, some consumers will remain idle until this balance equalises or tips in favour of the other side.

- **A consumer will commit the offset of a record when it is done processing it**. The commits are directed to a consumer coordinator, which will end up written to an internal `__consumer_offsets` topic. The offset of the record is incremented by one before committing, to prevent unnecessary replay.
- **Partitions may be manually assigned to free consumers**. If necessary, an entire topic may be assigned to a single free consumer — this is done by individually assigning all partitions.

Event streaming platforms are a highly effective building block in the construction of modular, loosely-coupled, event-driven applications. Within the world of event streaming, Kafka has solidified its position as the go-to open-source solution that is both flexible and highly performant. Concurrency and parallelism are at the heart of Kafka's architecture, forming partially-ordered event streams that can be load-balanced across a scalable consumer ecosystem. A simple reconfiguration of consumers and their encompassing groups can bring about vastly different event distribution and processing semantics; shifting the offset commit point can invert the delivery guarantee from an at-most-once to an at-least-once model.

The consumer group is a somewhat understated concept that is pivotal to the versatility of an event streaming platform. By simply varying the affinity of consumers with their groups, one can arrive at vastly different distribution topologies — from a topic-like, pub-sub behaviour to an MQ-style, point-to-point model. Because records are never truly consumed (the advancing offset only creates the illusion of consumption), one can concurrently superimpose disparate distribution topologies over a single event stream.

Chapter 4: Installation

Previous chapters have given us a reasonably grounded understanding of what Kafka is and isn't, where it is used, and how it dovetails into the rest of the software landscape. So far we have just been circling the shore; time to dive in for a deeper look. Before we can get much further, we need a running Kafka setup.

This will require us to install several things:

1. **Kafka and ZooKeeper**. Recall from our earlier coverage of the Kafka architecture, a functioning setup requires *both* Kafka and ZooKeeper nodes. These are plain Java applications with no other requirements or dependencies, and can run on any operating system and any hardware supported by Java.
2. **Kafdrop**. While this isn't strictly required to operate Kafka, Kafdrop is the most widely-used web-based tool for working with Kafka, and would be widely considered as an essential item of your Kafka toolkit.
3. **A Java Development Kit (JDK)**. Kafka is part-written in Scala and part in Java, and requires Java version 8 or newer to run. Kafdrop is somewhat more modern, requiring Java 11.

If you haven't done so already, install a copy of the JDK. Version 11 or newer will do. The rest of the chapter assumes that you have a JDK installed.

The requirement for JDK 11 is to accommodate Kafdrop. If you are installing Kafka on its own, JDK 8 will suffice. However, Java 11 is still recommended over Java 8 as it is the current *Long-Term Support* (LTS) version. The last free public updates of JDK 8 will have ended in January 2019. Free updates for JDK 11 will continue through to September 2021, by which point the world would have transitioned to JDK 17 LTS. (LTS releases are publicly supported for three years.)

Installing Kafka and ZooKeeper

There are at least four avenues at one's disposal for installing Kafka and ZooKeeper:

1. Run Kafka and ZooKeeper using Docker.
2. Install Kafka and ZooKeeper using a package manager, such as DNF (formerly known as YUM) for RedHat, CentOS and Fedora Linux distributions, APT for Debian and Ubuntu, and Homebrew for macOS. There are many others.

3. Clone the Kafka and ZooKeeper repositories and build from source code.
4. Download and unpack the official Kafka distribution from kafka.apache.org/downloads[1], which comes bundled with ZooKeeper.

Let's briefly touch upon each of these options. It might sound like overkill at first — we could just pick the easiest and get cracking — but we're in it for the long haul. And sometimes the easiest approach isn't the right one.

Docker is an excellent all-round approach for getting started with Kafka, developing against a local Kafka broker, and even running Kafka in a production configuration. And best of all, a Docker image will come with an appropriate version of the JDK.

There is one drawback, however. Kafka in Docker is notoriously difficult to configure, as everything is baked in and not designed for change. (That's not to say it can't be done.) We won't go down the Docker path for now, just because we will be making lots of changes to the broker configuration at various points along our journey, and ideally, we would want this process to be as simple and painless as possible.

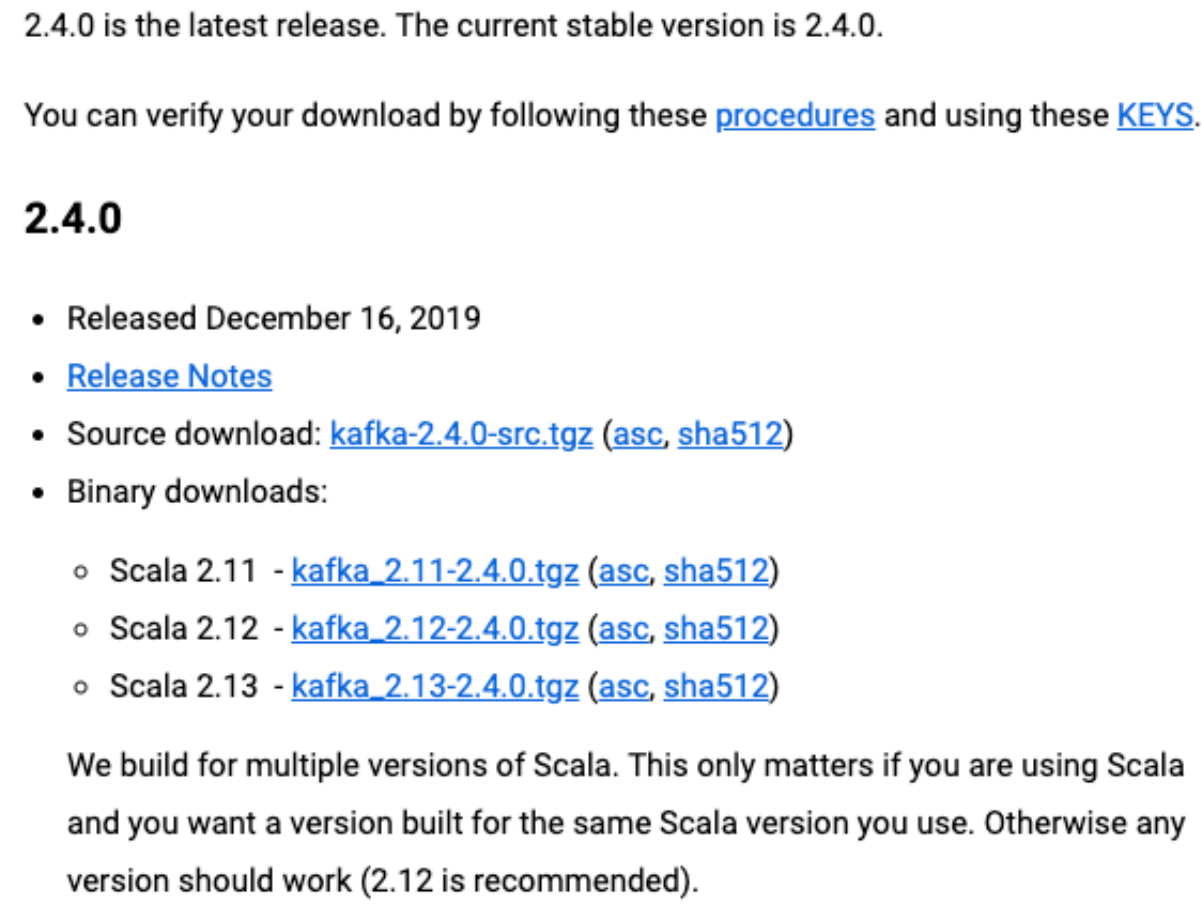
Installing Kafka and ZooKeeper from a package is convenient too. And while it doesn't come bundled with a JDK, a package will typically declare the JDK as a dependency, which the package manager will attempt to resolve at the point of installation (or when updating the package). Still, this approach isn't without its drawbacks: there may be other applications installed on the target machine that might require a different version of the JDK, which would warrant further configuration of Kafka and ZooKeeper to wire it up to the correct JDK.

Another drawback to the 'packaged Kafka' approach is that the installation path and the layout of the files will vary depending on the chosen package manager. For example, on macOS, Homebrew installs Kafka into `/usr/local/etc`, `/usr/local/bin`, and `/usr/local/var/lib/`. On the other hand, YUM will install it under `/bin`, `/opt`, and `/var/lib`. This makes it tremendously difficult to write about and include worked examples that work consistently for all readers. Rather than focusing on the subject matter, this book would have been polluted with excerpts and call-outs targeting different operating systems and package managers.

The third option is building from source code. It might sound a bit extreme for someone who has just opened a book on Kafka. Nonetheless, it is a valid option and the *only* option if you happen to be a contributor. Understandably, we won't delve into it much deeper, and blissfully pretend it doesn't exist — at least in the universe bound by this book.

The final option, and the one we will inevitably proceed with, is to download the latest version of the official Kafka tarball from kafka.apache.org/downloads[2]. There might be several options — pick the one in 'Binary downloads' that targets the latest version of Scala, as shown in the screenshot below.

[1] https://kafka.apache.org/downloads
[2] https://kafka.apache.org/downloads



2.4.0 is the latest release. The current stable version is 2.4.0.

You can verify your download by following these procedures and using these KEYS.

2.4.0

- Released December 16, 2019
- Release Notes
- Source download: kafka-2.4.0-src.tgz (asc, sha512)
- Binary downloads:
  - Scala 2.11 - kafka_2.11-2.4.0.tgz (asc, sha512)
  - Scala 2.12 - kafka_2.12-2.4.0.tgz (asc, sha512)
  - Scala 2.13 - kafka_2.13-2.4.0.tgz (asc, sha512)

We build for multiple versions of Scala. This only matters if you are using Scala and you want a version built for the same Scala version you use. Otherwise any version should work (2.12 is recommended).

Download options

Copy the downloaded `.tgz` file into a directory of your choice, and unpack with `tar zxf kafka_2.13-2.4.0.tgz`, replacing the filename as appropriate. (In this example, the downloaded version is 2.4.0, but your version will likely be newer.) The files will be unpacked to a subdirectory named `kafka_2.13-2.4.0`. We will refer to this directory as the **Kafka home directory**.

When referring to the home directory from a command-line example, we'll use the constant `$KAFKA_HOME`. You have the choice of either manually substituting `$KAFKA_HOME` for the installation directory, or assigning the installation path to the `KAFKA_HOME` environment variable, as shown in the example below.

```
export KAFKA_HOME=/Users/me/opt/kafka_2.13-2.4.0
```

You would need to run `export KAFKA_HOME...` at the beginning of every terminal session. Alternatively, you can append the `export ...` command to your shell's startup file. If you are using Bash, this is typically `~/.bashrc` or `~/.bash_profile`.

Take a moment to look around the home directory. You'll see several subdirectories, chief among them being `bin`, `libs`, and `config`.

```
.
├── LICENSE
├── NOTICE
├── bin
├── config
├── libs
└── site-docs
```

The `bin` directory contains the scripts to start and stop Kafka and ZooKeeper, as well as various CLI (command-line interface) utilities for working with Kafka. Also, `bin` contains a `windows` subdirectory, which (you've guessed it) contains the equivalent scripts for Microsoft Windows.

```
bin
├── connect-distributed.sh
├── connect-standalone.sh
├── kafka-acls.sh
├── kafka-broker-api-versions.sh
├── kafka-configs.sh
├── kafka-console-consumer.sh
├── kafka-console-producer.sh
├── kafka-consumer-groups.sh
├── kafka-consumer-perf-test.sh
├── kafka-delegation-tokens.sh
├── kafka-delete-records.sh
├── kafka-dump-log.sh
├── kafka-log-dirs.sh
├── kafka-mirror-maker.sh
├── kafka-preferred-replica-election.sh
├── kafka-producer-perf-test.sh
├── kafka-reassign-partitions.sh
├── kafka-replica-verification.sh
├── kafka-run-class.sh
├── kafka-server-start.sh
├── kafka-server-stop.sh
├── kafka-streams-application-reset.sh
├── kafka-topics.sh
├── kafka-verifiable-consumer.sh
├── kafka-verifiable-producer.sh
├── trogdor.sh
├── windows
```

```
├── zookeeper-security-migration.sh
├── zookeeper-server-start.sh
├── zookeeper-server-stop.sh
└── zookeeper-shell.sh
```

The `config` directory is another important one. It contains `.properties` files that are used to configure the various components that make up the Kafka ecosystem.

```
config
├── connect-console-sink.properties
├── connect-console-source.properties
├── connect-distributed.properties
├── connect-file-sink.properties
├── connect-file-source.properties
├── connect-log4j.properties
├── connect-standalone.properties
├── consumer.properties
├── log4j.properties
├── producer.properties
├── server.properties
├── tools-log4j.properties
├── trogdor.conf
└── zookeeper.properties
```

The `lib` directory contains the Kafka binary distribution, as well as its direct and transitive dependencies. You'll never need to modify the contents of this directory.

Launching Kafka and ZooKeeper

Now that the applications have been installed, we can start them. The first cab off the rank will be ZooKeeper, as it is a runtime requirement for Kafka. Run the following command in a terminal:

```
$KAFKA_HOME/bin/zookeeper-server-start.sh \
    $KAFKA_HOME/config/zookeeper.properties
```

This will launch ZooKeeper in foreground mode. You should see a bunch of messages logged to the console, signifying the starting of ZooKeeper. Among them, you might spot one warning message:

```
[2019-12-25 13:30:36,951] WARN Either no config or no quorum defined ␣
    in config, running in standalone mode (org.apache.zookeeper. ␣
    server.quorum.QuorumPeerMain)
```

All this is saying is that we started ZooKeeper in standalone mode, without configuring a quorum. When running ZooKeeper locally, availability is rarely a concern, and a standalone (ensemble of one member node) configuration is sufficient.

Recall from a prior discussion on the Kafka architecture, it was stated that ZooKeeper acts as an arbiter — electing a sole controller among the available Kafka broker nodes. Internally, ZooKeeper employs an atomic broadcast protocol to agree on and subsequently maintain a consistent view of the cluster state throughout the ZooKeepr *ensemble*. This protocol operates on the concept of a *majority vote*, also known as *quorum*, which in turn, requires an odd number of participating ZooKeeper nodes. When running in a production environment, ensure that at least three nodes are deployed in a manner that no pair of nodes may be impacted by the same contingency. Ideally, ZooKeeper nodes should be deployed in geographically separate data centres.

Now that ZooKeeper is running, we can start Kafka. Run the following in a new terminal window:

```
$KAFKA_HOME/bin/kafka-server-start.sh \
    $KAFKA_HOME/config/server.properties
```

Kafka's logs are a bit more verbose than ZooKeeper's. The first really useful part of the log relates to the ZooKeeper connection. Specifically, which ZooKeeper instance(s) Kafka is trying to connect to, and the status of the connection:

```
[2019-12-25 14:02:21,380] INFO Initiating client connection, ␣
    connectString=localhost:2181 sessionTimeout=6000 watcher= ␣
    kafka.zookeeper.ZooKeeperClient$ZooKeeperClientWatcher$@ ␣
    624ea235 (org.apache.zookeeper.ZooKeeper)
[2019-12-25 14:02:21,399] INFO [ZooKeeperClient Kafka server] ␣
    Waiting until connected. (kafka.zookeeper.ZooKeeperClient)
[2019-12-25 14:02:21,401] INFO Opening socket connection to server ␣
    localhost/0:0:0:0:0:0:0:1:2181. Will not attempt to ␣
    authenticate using SASL (unknown error) (org.apache.zookeeper. ␣
    ClientCnxn)
[2019-12-25 14:02:21,416] INFO Socket connection established to ␣
    localhost/0:0:0:0:0:0:0:1:2181, initiating session ␣
    (org.apache.zookeeper.ClientCnxn)
[2019-12-25 14:02:21,458] INFO Session establishment complete on ␣
    server localhost/0:0:0:0:0:0:0:1:2181, sessionid = ␣
```

```
    0x100433a40960000, negotiated timeout = 6000 □
    (org.apache.zookeeper.ClientCnxn)
[2019-12-25 14:02:21,461] INFO [ZooKeeperClient Kafka server] □
    Connected. (kafka.zookeeper.ZooKeeperClient)
```

Among the logs we can also find the complete Kafka broker configuration:

```
[2019-12-25 14:02:22,118] INFO KafkaConfig values:
  advertised.host.name = null
  advertised.listeners = null
  advertised.port = null
  alter.config.policy.class.name = null
  alter.log.dirs.replication.quota.window.num = 11
  alter.log.dirs.replication.quota.window.size.seconds = 1
  authorizer.class.name =
  auto.create.topics.enable = true
  auto.leader.rebalance.enable = true
  background.threads = 10
  broker.id = 0
  broker.id.generation.enable = true
  broker.rack = null
  client.quota.callback.class = null
  compression.type = producer
  connection.failed.authentication.delay.ms = 100
  ...
  (rest of the log omitted for brevity)
```

This is actually more useful than one might initially imagine. The Kafka broker configuration is defined in `$KAFKA_HOME/config/server.properties`, but the file is relatively small and initially contains mostly commented-out entries. This means that most settings are assigned their default values. Rather than consulting the official documentation to determine what the defaults might be and whether or not they are actually overridden in your configuration, you need only look at the broker logs. This is particularly useful when you need to debug the configuration. Suppose a particular configuration value isn't being applied correctly — perhaps due to a simple typo, or maybe because there are two entries for the same configuration key. The configuration printout in Kafka's logs provides its vantage point — as seen from the eyes of the broker.

The next useful bit of information is emitted by the socket listener:

```
[2019-12-25 14:02:22,587] INFO Awaiting socket connections on □
    0.0.0.0:9092. (kafka.network.Acceptor)
```

This tells us that Kafka is listening for inbound connections on port `9092`, and is bound to all network interfaces (indicated by the IP meta-address `0.0.0.0`). This is corroborated by the deprecated property `port`, which defaults to `9092`. There is a much more sophisticated mechanism for configuring listeners, which we will examine in one of the following chapters. For now, a `0.0.0.0:9092` listener will suffice.

Believe it or not, the most useful information one can get out of Kafka's logs is actually the version number. Admittedly, it sounds somewhat banal, but how many times have you stared helplessly at the screen wondering why a piece of software that was just upgraded to the latest version still has the same bug that the authors have sworn they had fixed? Invariably, it is always some simple mistake — a symlink to the wrong binary, a typo in the path, a wrong value in an environment variable, or some other moth-eaten stuff-up along those lines. Printing the application version number in the logs is a simple way of eradicating these classes of errors.

Running in the background

When launching ZooKeeper or Kafka, you have the option of running it as a daemon by passing it the `-daemon` flag. In simple terms, this means launching ZooKeeper in the background, without holding up the terminal. Kill the existing ZooKeeper process by pressing `CTRL+C`, and try the following:

```
$KAFKA_HOME/bin/zookeeper-server-start.sh -daemon \
    $KAFKA_HOME/config/zookeeper.properties
```

That's all well and good, but where have the logs gone? When launched as a daemon, the standard output of the ZooKeeper process is piped to `$KAFKA_HOME/logs/zookeeper.out`. We can tail the logs by running:

```
tail -f $KAFKA_HOME/logs/zookeeper.out
```

To stop a daemon ZooKeeper process, run `$KAFKA_HOME/bin/zookeeper-server-stop.sh`. This will stop the background process if there is one running. If not, it will respond with `No zookeeper server to stop`.

ZooKeeper and Kafka shell scripts are essentially mirror images of each other. To launch Kafka as a daemon, run:

```
$KAFKA_HOME/bin/kafka-server-start.sh -daemon \
    $KAFKA_HOME/config/server.properties
```

Kafka standard output logs are written to `$KAFKA_HOME/logs/kafkaServer.out`. To stop a daemon Kafka process, run `$KAFKA_HOME/bin/kafka-server-stop.sh`.

Installing Kafdrop

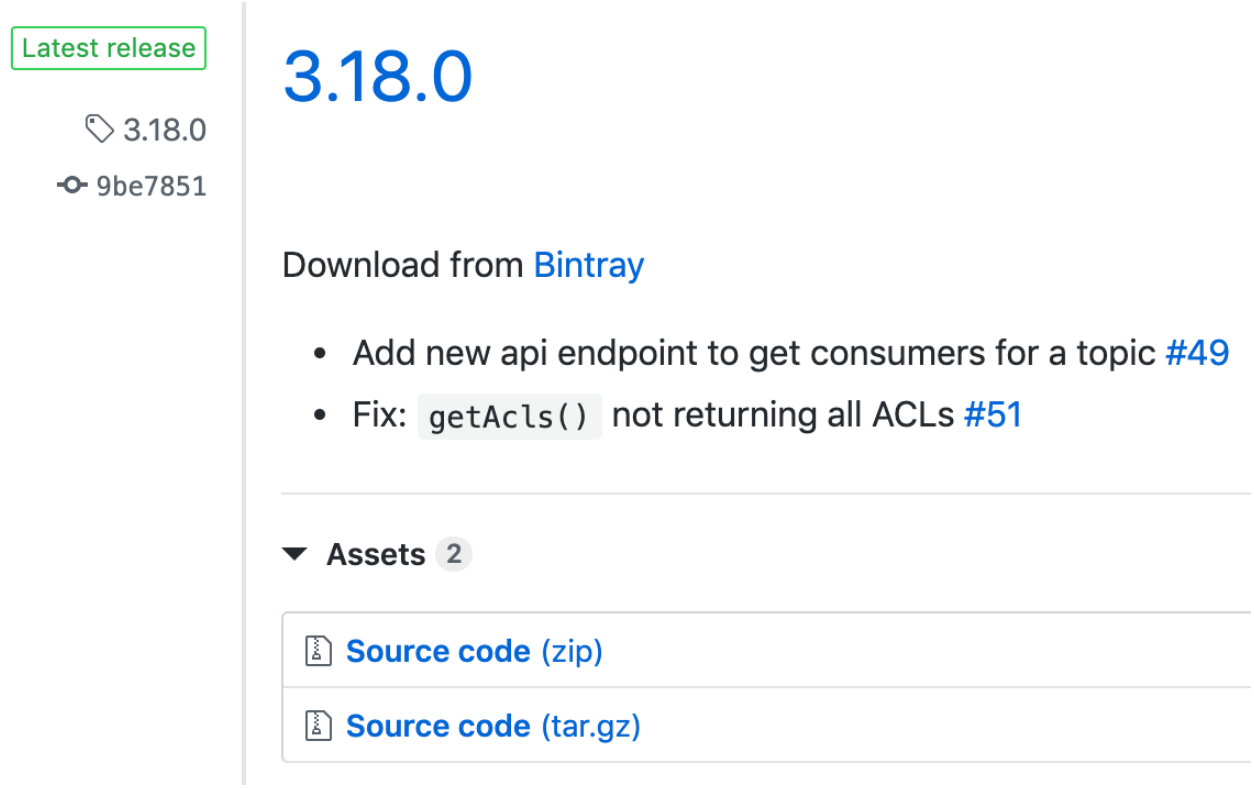
Next on our list is Kafdrop. It's a Java application with no dependencies, and the avenues for installing it are mostly similar to Kafka, except it does not offer package-based installation. In practice, Docker largely obviates the need for packages, and the sheer number of Kafdrop Docker pulls (over a million at the time of writing) is a testament to that.

As practical as a Docker image may be, we are going to ditch this option for now. Because we are running Kafka on `localhost`, Docker will struggle to connect to our broker, as Docker containers are normally unaware of processes running on the host machine. There is a way to change this but it is not portable across Linux and macOS, and will also require changes to the Kafka broker configuration — something we are not yet prepared to do. We will revisit Docker later. For now, we will go with the official Kafdrop binary distribution.

Kafdrop binaries are hosted on Bintray, with a download link embedded in each release on GitHub. Open the releases page: github.com/obsidiandynamics/kafdrop/releases[3] and pick the latest from the list. Alternatively, you can navigate straight to the latest Kafdrop release by following this shortcut: github.com/obsidiandynamics/kafdrop/releases/latest[4].

The release will typically include an outline of changes, and will contain a 'Download from Bintray' link, as shown in the example below:

[3] https://github.com/obsidiandynamics/kafdrop/releases
[4] https://github.com/obsidiandynamics/kafdrop/releases/latest



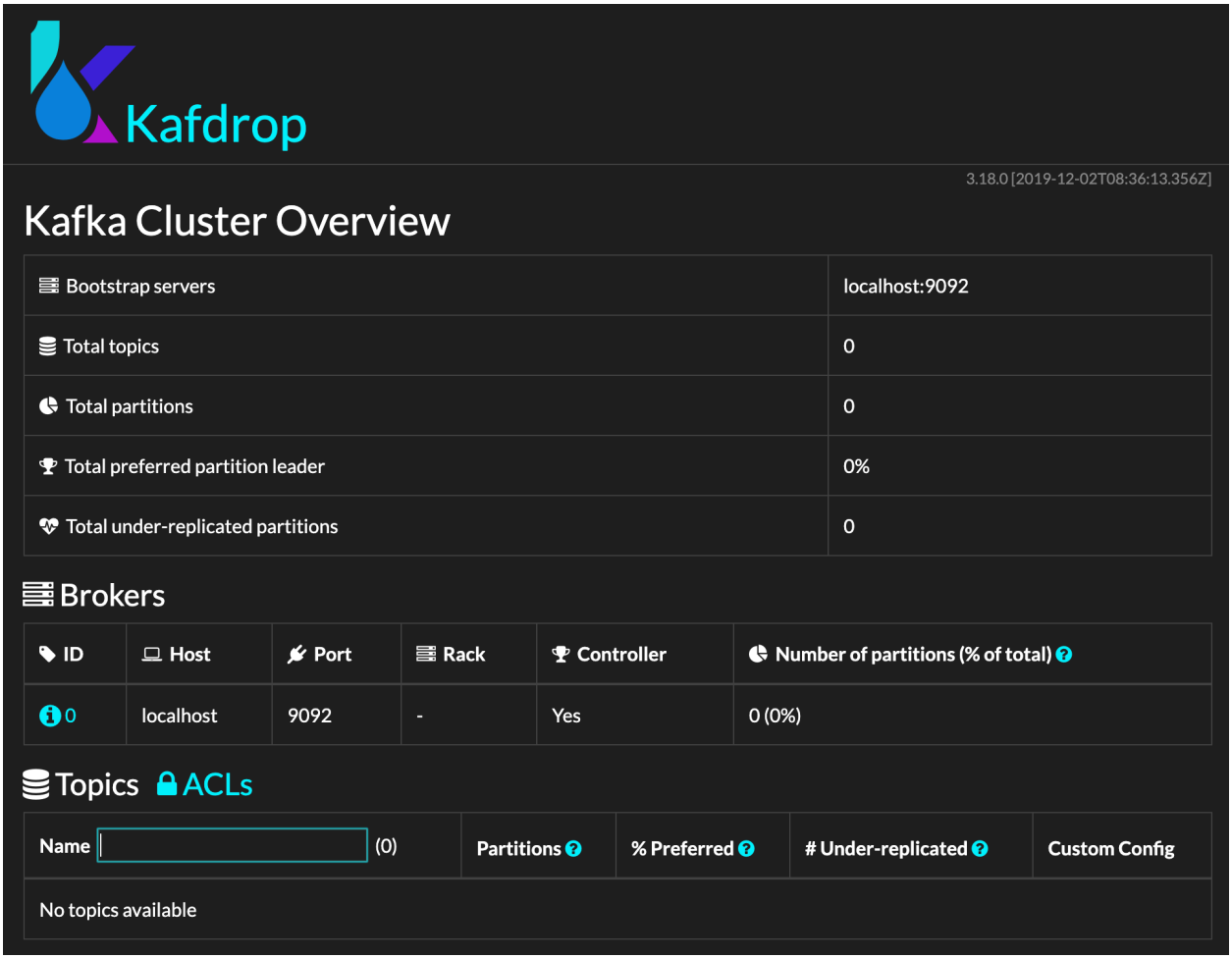
Download Kafdrop release

Clicking on the link will download a `.jar` file. Save it in a directory of your choice and run it as shown in the example below, replacing the filename as appropriate.

```
java -jar kafdrop-3.18.0.jar --kafka.brokerConnect=localhost:9092
```

Once started, you can open Kafdrop in your browser by navigating to localhost:9000[5]. You'll be presented with a Kafdrop cluster overview screen, showing our fresh, single-node Kafka cluster.

[5] http://localhost:9000



Kafdrop cluster overview

There are a few things of interest here. On the top-right corner, you should see the Kafdrop release version and buildstamp. This can be very useful if you are connecting to a remote Kafdrop instance, and don't have the logs that disclose which version of Kafdrop is running.

The next section provides a summary of the cluster. Note the 'Bootstrap servers' field: it mirrors the `--kafka.brokerConnect` command-line argument, telling us how Kafdrop has been configured to discover the Kafka nodes.

Bootstrapping and broker discovery is a whole topic on its own, which we are going to gloss over for now. For the time being, and unless stated otherwise, assume that the 'bootstrap servers' list is `localhost:9092`. We will revisit this topic in Chapter 8: Bootstrapping and Advertised Listeners.

The 'Brokers' section enumerates over the individual brokers in the cluster. We are running a single-broker setup just now, so seeing a one-line table entry should come as no surprise. Naturally, being the only broker in the cluster, it will have been assigned the controller role.

The 'Topics' section is empty, as we haven't created any topics yet. Nor do we have any ACLs defined. This is all yet to come.

Switch back to the shell running Kafdrop. Looking over the standard output logs we can spot the version number. Keep looking and you'll notice the port that Kafdrop is listening on and the context path. This is all configuration, and it's something that may need to change between environments.

```
2019-12-25 19:08:49.465 INFO 82515 [main] k.s.BuildInfo: Kafdrop 
    version: 3.18.0, build time: 2019-12-02T08:36:13.356Z
...
(some logs omitted)
...
2019-12-25 19:08:50.752 INFO 82515 [main] o.s.b.w.e.u. 
    UndertowServletWebServer: Undertow started on port(s) 
    9000 (http) with context path ''
```

We have learned about the various ways one can obtain and install a Kafka and ZooKeeper bundle. We have also started and stopped a basic ZooKeeper and Kafka setup, learned about foreground and daemon modes, and surveyed the logs for useful information. Finally, we installed Kafdrop and took a brief look around. The scene is now set; we just need to make use of it all somehow.

Chapter 5: Getting Started

With the theoretical foundations nailed, and a fresh installation of Kafka standing by, it is time to roll up our sleeves for a more practical approach to learning Kafka.

This chapter will focus on the two fundamental operations: publishing records to Kafka topics and subsequently consuming them. We are going to explore the various mechanisms for interacting with the broker and also for exploring the contents of topics and partitions.

Publishing and consuming using the CLI

When discussing producers and consumers, the first thing that might spring to mind is a set of bespoke applications that *someone* — an individual, or more likely, a team of developers — will build and maintain as part of operating a broader event-streaming system. But one does not need a fully-fledged application to publish to or consume from a Kafka topic — this task can be accomplished using the set of CLI (command-line interface) tools that are shipped with Kafka, located in the `$KAFKA_HOME/bin` directory.

Creating a topic

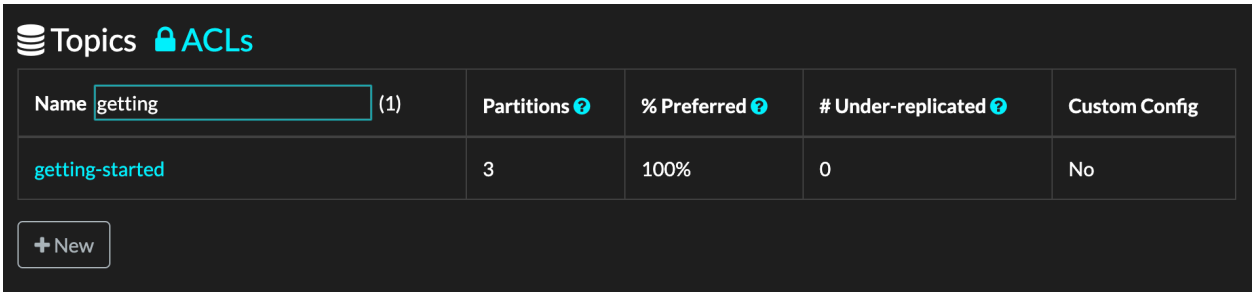
Let's get started then. The first thing is to create a topic, which can be accomplished using the `kafka-topics.sh` tool:

```
$KAFKA_HOME/bin/kafka-topics.sh --bootstrap-server localhost:9092 \
    --create --partitions 3 --replication-factor 1 \
    --topic getting-started
```

Observe, although the parameter `--bootstrap-server` is named in singular form, the `kafka-topics.sh` tool will, rather unexpectedly, accept a comma-separated list of brokers. We have specified `localhost:9092` as the bootstrap server, because that is where our test cluster is currently running. If you are using a remote Kafka broker or a managed Kafka service, you will have been provided with an alternate list of broker addresses.

The packaged CLI utilities are not the most intuitive of tools that one can use with Kafka; in fact, they are widely regarded as being awkward to use and barely adequate in functionality. Most Kafka practitioners have long abandoned the out-of-the-box utilities in favour of other open-source and commercial tools; Kafdrop is one such tool, but there are several others. This book covers the packaged CLI tools because that is what you are sure to get with every Kafka installation. Having basic awareness of the built-in tooling is about as essential as knowing the basic `vi` commands when working in Linux — you can berate the archaic tooling and laud the alternatives, but that will only get you so far as your first production incident. (In saying that, comparing Kafka's built-in tooling to Vim is a travesty.)

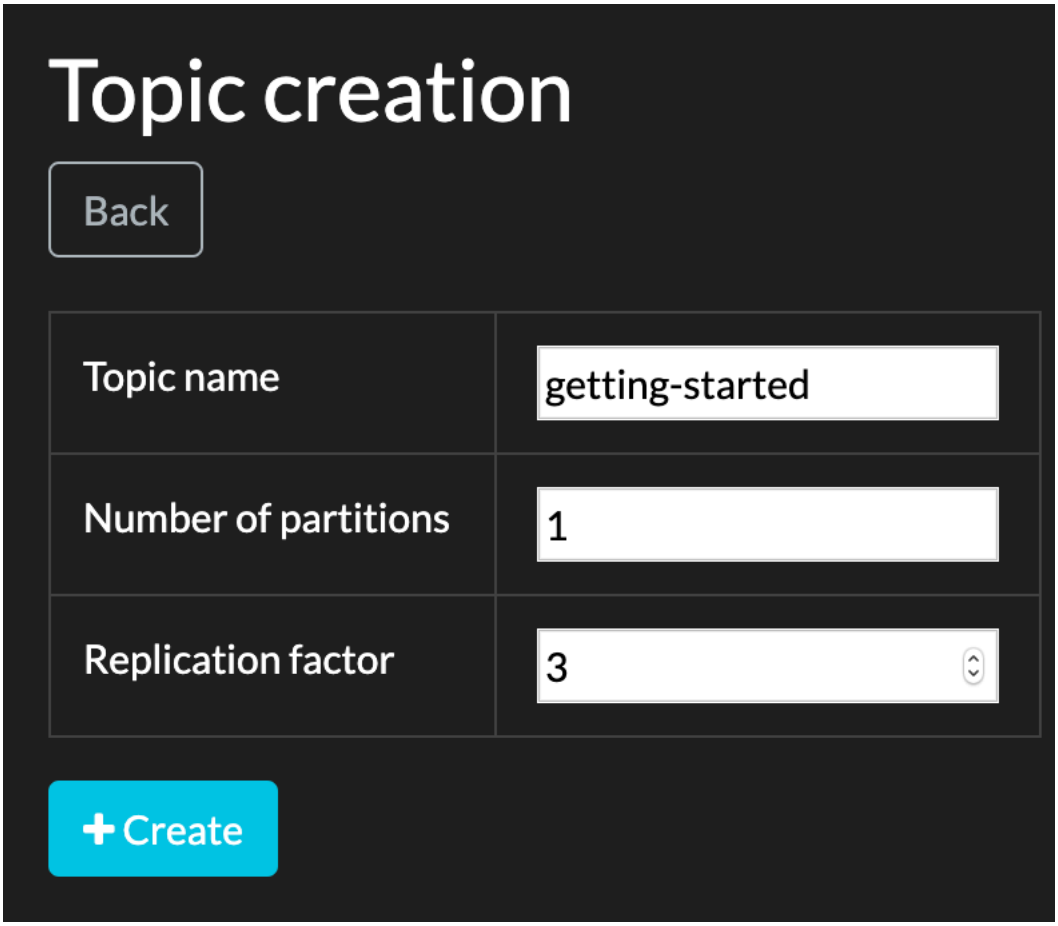
Switching to Kafdrop, we can see the `getting-started` topic appear in the 'Topics' section. If there are lots of topics, you can use the filter text box in the table area to refine the displayed topics to just those that match a chosen substring.



Kafdrop — showing topics

We can tell at a glance that the topic has three partitions, the topic has no replication issues, and that no custom configuration has been specified for this topic. The mechanics for specifying per-topic configuration will be explained in Chapter 9: Broker Configuration.

Now, we could have just as easily created a topic by clicking the 'New' button under the topics list:



Kafdrop — creating a new topic

However, the point of the exercise is to demonstrate the CLI tool, rather than to explore all the possible ways one can create a topic in Kafka.

> This might be a good segue to discuss the importance of explicit topic creation. Kafka does not require clients to create topics by default. When the `auto.create.topics.enable` broker configuration property is set to `true`, Kafka will automatically create the topic when clients attempt to produce, consume, or fetch metadata for a non-existent topic. This might sound like a nifty idea at first, but the drawbacks significantly outweigh the minor convenience one gets from not having to create the topic explicitly.
>
> Firstly, Kafka's defaults to back as far as 2011, and are generally optimised for the sorts of use cases that Kafka was originally designed for — high volume log shipping. Many things have changed since; as it happens, Kafka is no longer a one-trick pony. As such, one would typically want to configure the topic at the point of creation, or immediately thereafter — certainly before it gets a real workout.
>
> Secondly, the partition count: Kafka allows you to specify the default number of partitions for all newly created topics using the `num.partitions` broker setting, but this is largely a meaningless number. Topics should be sized individually on the basis of expected parallelism, and a number

of other factors, which are discussed in Chapter 6: Design Considerations. Specifying the partition count requires explicit topic creation. A similar statement might be made regarding the replication factor, but it is arguably easier to agree on a sensible default for the replication factor than it is for the partition count.

Finally, having Kafka auto-create topics when a client subscribes to a topic or simply fetches the topic metadata is careless, to put it mildly. A misbehaving client may initiate arbitrary metadata queries that could inadvertently create a copious number of stray topics.

Publishing records

With the topic creation out of the way, let's publish a few records. We are going to use the `kafka-console-producer.sh` tool:

```
$KAFKA_HOME/bin/kafka-console-producer.sh \
    --broker-list localhost:9092 \
    --topic getting-started --property "parse.key=true" \
    --property "key.separator=:"
```

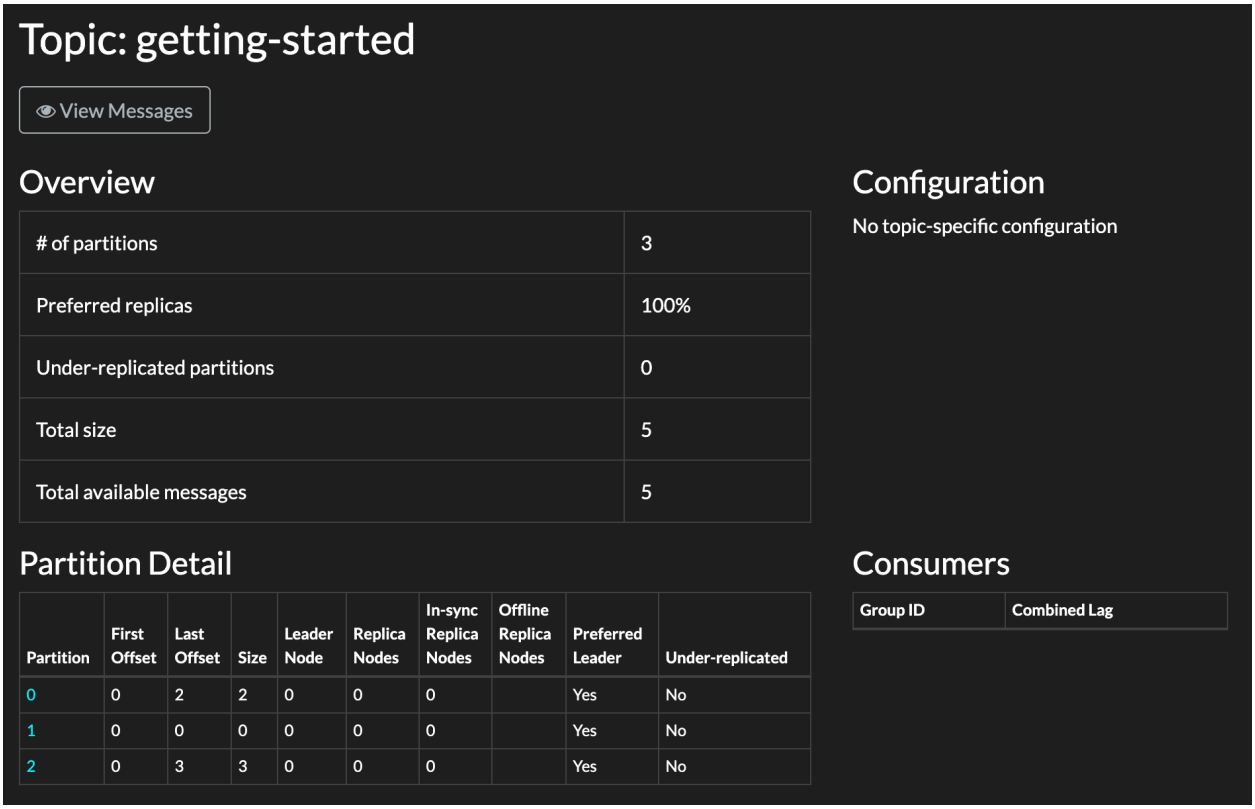
Records are separated by newlines. The key and the value parts are delimited by colons, as indicated by the `key.separator` property. For the sake of an example, type in the following (a copy-paste will do):

```
foo:first message
foo:second message
bar:first message
foo:third message
bar:second message
```

Press `CTRL+D` when done. The terminal echoes a right angle bracket (`>`) for every record published.

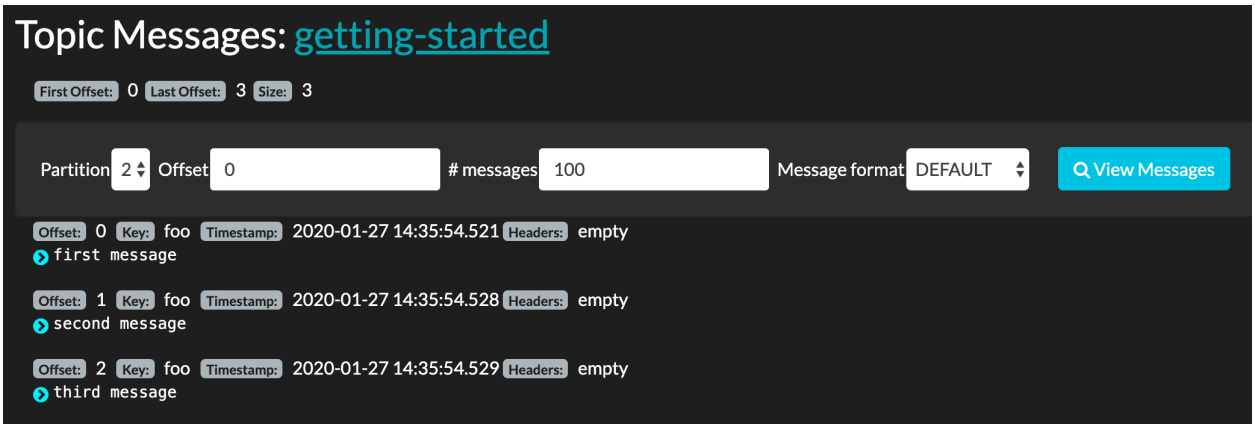
Note, the `kafka-topics.sh` tool uses the `--bootstrap-server` parameter to configure the Kafka broker list, while `kafka-console-producer.sh` uses the `--broker-list` parameter for an identical purpose. Also, `--property` arguments are largely undocumented — be prepared to Google your way around.

At this point we can switch back to Kafdrop and view the contents of the `getting-started` topic. We are presented with an overview of the topic, along with a detailed breakdown of the underlying partitions:



Kafdrop — view topic contents

Focusing on the partition detail, we can tell at a glance that of the three partitions, two have data and one is empty. The 'first offset' and 'last offset' columns correspond to the low-water and high-water marks, respectively. As the reader might recall from Chapter 3: Architecture and Core Concepts, subtracting the two yields the maximum number of records persisted in the partition. Let's click on partition #2. Kafdrop will show the individual records, arranged in chronological order.



Kafdrop — view partition contents

In case you were wondering, the arrow to the left of the record lets you expand and pretty-print JSON-encoded records. As our examples didn't use JSON, there's nothing to pretty-print.

Consuming records

```
$KAFKA_HOME/bin/kafka-console-consumer.sh \
    --bootstrap-server localhost:9092 \
    --topic getting-started --group cli-consumer --from-beginning \
    --property "print.key=true" --property "key.separator=:"
```

The terminal will echo the following:

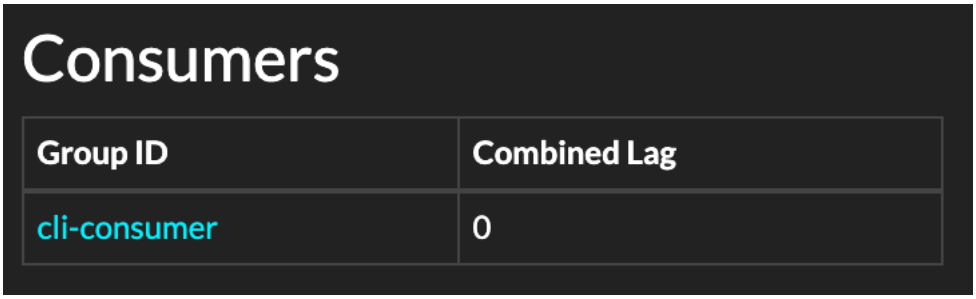
```
bar:first message
bar:second message
foo:first message
foo:second message
foo:third message
```

Because the consumer is running as a subscription, with a provided consumer group, the output will stall on the last record. The consumer will effectively tail the topic — continuously polling for new records and printing them as they arrive on the topic. To terminate the consumer, press `CTRL+D`.

Note that we specified the `--from-beginning` flag when invoking the command above. By default, a first-time consumer (for a previously non-existent group) will have its offsets reset to the topic's high-water mark. In order to read the previously published records, we override the default offset reset strategy to tail from the topic's low-water mark. If we run the same command again, we will see no records — the consumer will halt, waiting for the arrival of new records.

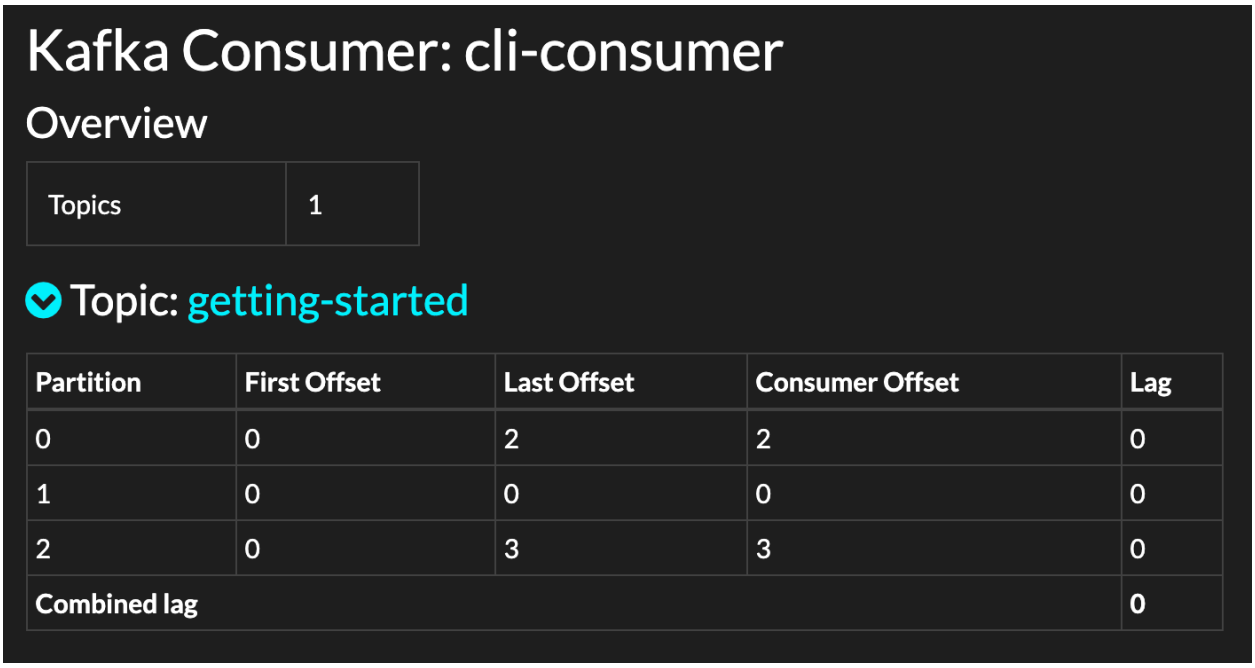
There is no `--from-end` flag. To tail from the end of the topic, simply delete the consumer offsets and start the CLI consumer. Deleting offsets and other offset manipulation commands are described in the section that follows.

Having consumed the backlog of records with the new `cli-consumer` consumer group, we can now switch back to Kafdrop to observe the addition of the new group. The new group appears in the topic overview screen, under the section 'Consumers', in the bottom-right.



Kafdrop — consumer groups for a topic

Clicking through the consumer link takes us to the consumer overview. This screen enumerates over all topics within the consumer's subscription, as well as the per-partition offsets for each topic.



Kafdrop — consumer overview

In our example, the consumer offset recorded for each partition is the same as the respective high-water mark. The consumer lag is zero for each column. This is the difference between the committed offset and the high-water mark. When the lag is zero, it means that the consumer has worked through the entire backlog of records for the partition; in other words, the consumer has caught up to the producer. Lag may vary between partitions — the busier the partition, in terms of record throughput, the more likely it will accumulate lag. The aggregate lag (also known as the combined lag) is the sum of all individual per-partition lags.

Among the useful characteristics of tools such as Kafdrop and the Kafka CLI is the ability to enumerate and monitor individual consumer groups — inspect the per-partition lags and spot leading indicators of degraded consumer performance or, in the worst-case scenario, a stalled consumer. Much like any other middleware, a solid comprehension of the available tooling — be it the built-in suite or the external tools — is essential to effective operation. This is particularly crucial for overseeing mission-critical systems in production environments, where minutes of downtime and the fruitless head-scratching of the engineering and support personnel can result in significant incurred losses.

So there you have it. We have published and consumed records from a Kafka topic using the built-in CLI tools. It isn't much, but it's a start.

Useful CLI commands

To close off the section on the CLI, we will take a brief look at the other useful actions that can be performed using the built-in tools.

Listing topics

The `kafka-topics.sh` tool can be used to list topics, as per the example below.

```
$KAFKA_HOME/bin/kafka-topics.sh \
    --bootstrap-server localhost:9092 \
    --list --exclude-internal
```

The `--exclude-internal` flag, as the name suggests, eliminates the internal topics (e.g. `__consumer_offsets`) from the query results.

Describing a topic

By passing the `--describe` flag and a topic name to `kafka-topics.sh`, we can get more detailed information about a specific topic, including the partition leaders, follower replicas, and the in-sync replica set:

```
$KAFKA_HOME/bin/kafka-topics.sh \
    --bootstrap-server localhost:9092 \
    --describe --topic getting-started
```

Produces:

```
Topic: getting-started  PartitionCount: 3 ReplicationFactor: 1 □
    Configs: segment.bytes=1073741824
  Topic: getting-started  Partition: 0  Leader: 0 Replicas: 0 Isr: 0
  Topic: getting-started  Partition: 1  Leader: 0 Replicas: 0 Isr: 0
  Topic: getting-started  Partition: 2  Leader: 0 Replicas: 0 Isr: 0
```

Deleting a topic

To delete an existing topic, use the `kafka-topics.sh` tool. The example below deletes the `getting-started` topic from our test cluster.

```
$KAFKA_HOME/bin/kafka-topics.sh \
    --bootstrap-server localhost:9092 \
    --topic getting-started --delete
```

Topic deletion is an asynchronous operation — a topic is initially marked for deletion, to be subsequently cleaned up by a background process at an indeterminate time in the future. In-between the marking and the final deletion, a topic might appear to linger around — only to disappear moments later.

The asynchronous behaviour of topic deletion should be taken into account when dealing with short-lived topics — for example, when conducting an integration test. The latter typically requires a state reset between successive runs, wiping associated database tables and event streams. Because there is no equivalent of a blocking `DELETE TABLE` DDL operation in Kafka, one must think outside the box. The options are:

1. Forcibly reset consumer offsets to the high-water mark prior to each test, delete the offsets, or delete the consumer group (all three will achieve equivalent results);
2. Truncate the underlying partitions by shifting the low-water mark (truncation will be described shortly); or
3. Use unique, disposable topic names for each test, deleting any ephemeral topics when the test ends.

The latter is the recommended option, as it creates due isolation between tests and allows multiple tests to operate concurrently with no mutually-observable side-effects.

Truncating partitions

Although a partition is backed by an immutable log, Kafka offers a mechanism to truncate all records in the log up to a user-specified low-water mark. This can be achieved by passing a JSON document to the `kafka-delete-records.sh` tool, specifying the topics and partitions for truncation, with the new low-water mark in the `offset` attribute. Several topic-partition-offset triples can be specified as a batch. In the example below, we are truncating the first record from `getting-started:2`, leaving records at offset 1 and newer intact.

```
cat << EOF > /tmp/offsets.json
{
  "partitions": [
    {"topic": "getting-started", "partition": 2, "offset": 1}
  ],
  "version": 1
}
EOF
$KAFKA_HOME/bin/kafka-delete-records.sh \
    --bootstrap-server localhost:9092 \
    --offset-json-file /tmp/offsets.json
```

In an analogous manner, we can truncate the entire partition by specifying the current high-water mark in the `offset` attribute.

Listing consumer groups

The `kafka-consumer-groups.sh` tool can be used to query Kafka for a list of consumer groups.

```
$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 --list
```

The result is a newline-separated list of consumer group names. This output format conveniently allows us to iterate over groups, enacting repetitive group-related administrative operations from a shell script.

```
#!/bin/bash

list_groups_cmd="$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 --list"
for group in $(bash -c $list_groups_cmd); do
  # do something with the $group variable
done
```

Describing a consumer group

The same tool can be used to display detailed state information about each consumer group — namely, its partition offsets for the set of subscribed topics. A sample invocation and the resulting output is shown below.

```
$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --group cli-consumer --describe --all-topics
```

Produces the following when no consumers are connected:

```
Consumer group 'cli-consumer' has no active members.

GROUP         TOPIC           PARTITION  CURRENT-OFFSET
cli-consumer  getting-started 1          0
cli-consumer  getting-started 0          2
cli-consumer  getting-started 2          3
 □
LOG-END-OFFSET  LAG  CONSUMER-ID  HOST  CLIENT-ID
0               0    -            -     -
2               0    -            -     -
3               0    -            -     -
```

If, on the other hand, we attach a consumer (from an earlier example, using the `kafka-console-consumer.sh` tool), the output resembles the following:

```
GROUP           TOPIC           PARTITION  CURRENT-OFFSET
cli-consumer    getting-started 0          2
cli-consumer    getting-started 1          0
cli-consumer    getting-started 2          3
 □
LOG-END-OFFSET  LAG
2               0
0               0
3               0
 □
CONSUMER-ID
consumer-cli-consumer-1-077c1bf1-df64-4d3e-a479-350e962119cc
consumer-cli-consumer-1-077c1bf1-df64-4d3e-a479-350e962119cc
consumer-cli-consumer-1-077c1bf1-df64-4d3e-a479-350e962119cc
 □
HOST            CLIENT-ID
/127.0.0.1      consumer-cli-consumer-1
/127.0.0.1      consumer-cli-consumer-1
/127.0.0.1      consumer-cli-consumer-1
```

In addition to describing a specific consumer group, this tool can be used to describe all groups:

```
$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --describe --all-groups --all-topics
```

The `--describe` flag has an complementary flag — `--state` — that drills into the present state of the consumer group. This includes the ID of the coordinator node, the assignment strategy, the number of active members, and the state of the group. These attributes are explained in greater detail in Chapter 15: Group Membership and Partition Assignment. The example below illustrates this command and its sample output.

```
$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --describe --all-groups --state


GROUP                COORDINATOR (ID)
cli-consumer         localhost:9092 (0)
 □
ASSIGNMENT-STRATEGY  STATE           #MEMBERS
range                Stable          1
```

Resetting offsets

In the course of working with Kafka, we occasionally come across a situation where the committed offsets of a consumer group require minor adjustment; in the more extreme case, that adjustment might entail a complete reset of the offsets. An adjustment might be necessary if, for example, the consumer has to skip over some records — perhaps due to the records containing erroneous data. (These are sometimes referred to as 'poisoned' records.) Alternatively, the consumer may be required to reprocess earlier records — possibly due to a bug in the application which was subsequently resolved. Whichever the reason, the `kafka-consumer-groups.sh` tool can be used with the `--reset-offsets` flag to affect fine-grained control over the consumer group's committed offsets.

The example below rewinds the offsets for the consumer group `cli-consumer` to the low-water mark, using the `--to-earliest` flag — resulting in the forced reprocessing of all records when the consumer group reconnects. Alternatively, the `--to-latest` flag can be used to fast-forward the offsets to the high-water mark extremity, skipping all backlogged records. Resetting offsets is an offline operation; the operation will not proceed in the presence of a connected consumer.

```
$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --topic getting-started --group cli-consumer \
    --reset-offsets --to-earliest --execute
```

By default, passing the `--reset-offsets` flag will result in a *dry run*, whereby the tool will list the partitions that will be subject to a reset, the existing offsets, as well as the candidate offsets that will be assigned upon completion. This is equivalent of running the tool with the `--dry-run` flag, and is designed to protect the user from accidentally corrupting the consumer group's state. To enact the change, run the command with the `--execute` flag, as shown in the example above.

In addition to resetting offsets for the entire topic, the reset operation can be performed selectively on a subset of the topic's partitions. This can be accomplished by passing in a list of partition numbers following the topic name, in the form `<topic-name>:<first-partition>,<second-partition>,...,<N-th-partitio`. An example of this syntax is featured below. Also, rather than resetting the offset to a partition extremity, this example uses the `--to-offset` parameter to specify a numeric offset.

```
$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --topic getting-started:0,1 --group cli-consumer \
    --reset-offsets --to-offset 2 --execute
```

The next example uses Kafka's record time-stamping to locate an offset based on the given date-time value, quoted in ISO 8601 form. Specifically, the offsets will be reset to the earliest point in time that occurs at the specified timestamp or after it. This feature is convenient when one needs to wind the offsets back to a known point in time. When using the `--to-datetime` parameter, ensure that the offset is passed using the correct timezone; if unspecified, the timezone defaults to the Coordinated Universal Time (UTC), also known as Zulu time. In the example below, the timezone had to be adjusted to Australian Eastern Daylight Time (AEDT), eleven hours east of Zulu, as this book was written in Sydney.

```
$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --topic getting-started:2 --group cli-consumer \
    --reset-offsets --to-datetime 2020-01-27T14:35:54.528+11:00 \
    --execute
```

The final option offered by this tool is to shift the offsets by a fixed quantity *n*, using the `--shift-by` parameter. The magnitude of the shift may be a positive number — for a forward movement, or a negative number — to rewind the offsets. The extent of the shift is bounded by the partition extremities; the result of 'current offset' + *n* will be capped by the low-water and high-water marks.

Deleting offsets

Another method of resetting the offsets is to delete the offsets altogether, shown in the example below. This is, in effect, a lazy form of reset — the assignment of new offsets does not occur until a consumer connects to the cluster. When this happens, the `auto.offset.reset` client property will stipulate which extremity the offset should be reset to — either the earliest offset or the latest.

```
$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --topic getting-started --group cli-consumer --delete-offsets
```

Deleting a consumer group

Deleting a consumer group erases all persistent state associated with it. This is accomplished by passing the `--delete` flag to the `kafka-consumer-groups.sh` CLI, as shown below.

```
$KAFKA_HOME/bin/kafka-consumer-groups.sh \
    --bootstrap-server localhost:9092 \
    --group cli-consumer --delete
```

Deleting the consumer group is equivalent to deleting offsets for all topics and all partitions.

A basic Java producer and consumer

A CLI is a great place to start, and can be driven programmatically from a shell script. But this is more of a convenience; an automation of repetitive functionality, if you will. Any serious event streaming application will employ a high-level language such as Java, C, or Python to implement the business logic required to publish records and to react to events emitted by other applications.

Client libraries

Unlike the built-in CLI, which relies on the presence of binaries and a pre-installed Java runtime, applications rely solely on distributable client libraries. These are available for just about every programming language under the sun, from the mainstream to the esoteric.

In this book, we are going to focus solely on the Java ecosystem — being among the most popular mainstream software development environments and the 'home turf' of Kafka and many related event streaming technologies. The Java client implementation is the most mature of the available client libraries, being developed alongside and at the same cadence as the Kafka broker. Other languages will have similar clients; they are maintained independently of Kafka and feature varying level of feature support and stability. Bear in mind, these libraries will slightly lag the mainstream Kafka releases in terms of feature sets; if you are after 'bleeding edge' capabilities, you will be best served by the Java client library and, to a marginally lesser extent, the C library — `librdkafka`, maintained by Magnus Edenhill.

Using the Java library

To add a Kafka client library to your project, add the following to your `build.gradle` (if using Gradle):

```
dependencies {
    implementation "org.apache.kafka:kafka-clients:2.4.0"
}
```

Alternatively, if using Maven, add the following to your `pom.xml`:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.4.0</version>
</dependency>
```

The examples above assume Kafka version 2.4.0 — the latest at the time of writing. Replace this with a more up-to-date version if appropriate.

The complete source code for the upcoming examples is available at github.com/ekoutanov/effectivekafka[6] in the `src/main/java/effectivekafka/basic` directory. Code listings will have their package declaration removed for brevity, and often will strip out `import` statements and outer class declarations.

Interfacing with the Kafka client libraries is done primarily using the following classes:

- `Producer`: The public interface of the producer client, containing the necessary method signatures for publishing records and using transactions. This interface is surprisingly light on documentation; method comments simply delegate the documentation to the concrete implementation.
- `KafkaProducer`: The implementation of `Producer`. In addition, a `KafkaProducer` contains detailed Javadoc comments for each method.
- `ProducerRecord`: A data structure encompassing the attributes of a record, as perceived by a producer. To be precise, this is the representation of a record *before* it has been published to a partition; as such, it contains only the basic set of attributes: topic name, partition number, optional headers, key, value, and a timestamp.
- `Consumer`: The definition of a consumer entity, containing message signatures for controlling subscriptions and topic/partition assignment, fetching records from the cluster, committing offsets, and obtaining information about the available topics and partitions.

[6] https://github.com/ekoutanov/effectivekafka/tree/master/src/main/java/effectivekafka/basic

- `KafkaConsumer`: The implementation of `Consumer`. Like its producer counterpart, this implementation contains the complete set of Javadocs.
- `ConsumerRecord`: A consumer-centric structure for housing record attributes. A `ConsumerRecord` is effectively a superset of the `ProducerRecord`, containing additional metadata such as the record offset, the checksum, and some other internal attributes.

There are other classes that are used, from time to time, to interface with the client library. However, the bulk of record publishing and consumption can be achieved using little more than just the six classes above.

Publishing records

A simple, yet complete example illustrating the publishing of Kafka records is presented below.

```
import static java.lang.System.*;

import java.util.*;

import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.*;

public final class BasicProducerSample {
  public static void main(String[] args)
      throws InterruptedException {
    final var topic = "getting-started";

    final Map<String, Object> config =
        Map.of(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9092",
               ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
               StringSerializer.class.getName(),
               ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
               StringSerializer.class.getName(),
               ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,
               true);

    try (var producer = new KafkaProducer<String, String>(config)) {
      while (true) {
        final var key = "myKey";
        final var value = new Date().toString();
        out.format("Publishing record with value %s%n",
                   value);
```

```
        final Callback callback = (metadata, exception) -> {
          out.format("Published with metadata: %s, error: %s%n",
                     metadata, exception);
        };

        // publish the record, handling the metadata in the callback
        producer.send(new ProducerRecord<>(topic, key, value),
                      callback);

        // wait a second before publishing another
        Thread.sleep(1000);
      }
    }
  }
}
```

The first item on our to-do list is to configure the client. This is done by building a mapping of property names to configured values. More detailed information on configuration is presented in Chapter 10: Client Configuration; for the time being, we will limit ourselves to the most basic configuration options — just enough to get us going with a functioning producer.

The configuration keys are strings — being among the permissible property names defined in the official Kafka documentation, available online at kafka.apache.org/documentation[7]. Rather than quoting strings directly, our example employs the static constants defined in the `ProducerConfig` class, thereby avoiding a mistype.

Of the four configuration mappings supplied, the first specifies a list of so-called *bootstrap servers*. In our example, this is a singleton list comprising the endpoint `localhost:9092` — the address of our test broker. Bootstrapping is a moderately involved topic, described in Chapter 8: Bootstrapping and Advertised Listeners.

The next two mappings specify the serializers that the producer should use for the records' keys and values. Kafka offers lots of options around how keys and values are marshalled — using either built-in or custom serializers. For the sake of expediency, we will go with the simplest option at our disposal — writing records as plain strings. More elaborate forms of marshalling will be explored in Chapter 7: Serialization.

Whereas the first three items represent mandatory configuration, the fourth is entirely optional. By default, in the absence of idempotence, a producer may inadvertently publish a record in duplicate or out-of-order — if one of the queued records experiences a timeout during publishing and is reattempted after one or more of its successors have gone through. With the `enable.idempotence` option set to `true`, the broker will maintain an internal sequence number for each producer and

[7] https://kafka.apache.org/documentation/#producerconfigs

partition pair, ensuring that records are not processed in duplicate or out-of-order. So it's good practice to enable idempotence.

Prior to publishing a record, we need to instantiate a `KafkaProducer`, giving it the assembled config map in the constructor. A producer cannot be reconfigured following instantiation. One instantiated, we will keep a reference to the `KafkaProducer` instance, as it must be closed when the application no longer needs it. This is important because a `KafkaProducer` maintains TCP connections to multiple brokers and also operates a background I/O thread to ferry the records across. Failure to close the producer instance may result in resource starvation on the client, as well as on the brokers. As `Producer` extends the `Closeable` interface, the best way to ensure that the producer instance is properly disposed of is to use a *try-with-resources* block, as shown in the listing above.

Sometimes we need a producer to hang around indefinitely — for example, when an application publishes events in response to some external stimuli, such as responding to an API request. In this scenario, the use of a *try-with-resources* is inappropriate, as the lifecycle of a `KafkaProducer` instance is obviously aligned with that of the API controller or the associated business logic layer (depending on how the application is architected). Instead, we would let the owner of the producer, whichever component that may be, deal with lifecycle concerns.

In order to actually publish a record, one must use the `Producer.send()` API. There are two overloaded variations of `send()` method:

1. `Future<RecordMetadata> send(ProducerRecord<K, V> record)`: asynchronously sends the record, returning a `Future` containing the record metadata.
2. `Future<RecordMetadata> send(ProducerRecord<K, V> record, Callback callback)`: asynchronously sends the record, invoking the given `Callback` implementation when either the record has been successfully persisted on the broker or an error has occurred. Our example uses this variant.

The `send()` methods are asynchronous, returning as soon as the record is serialized and staged in the accumulator buffer. The actual sending of the record will be performed in the background, by a dedicated I/O thread. To block on the result, the application can invoke the `get()` method of the provided `Future`.

The publishing of records takes place in a loop, with a one second sleep between each successive `send()` call. For simplicity, we are publishing the current date, keyed to a constant `"myKey"`. This means that all records will appear on the same partition.

Running the example above results in the following output (until terminated):

```
13:14:09/0  INFO  [main]: [Producer clientId=basic-producer-sample] ▯
    Instantiated an idempotent producer.
13:14:09/66 INFO  [main]: [Producer clientId=basic-producer-sample] ▯
    Overriding the default retries config to the recommended ▯
    value of 2147483647 since the idempotent producer is ▯
    enabled.
13:14:09/66 INFO  [main]: [Producer clientId=basic-producer-sample] ▯
    Overriding the default acks to all since idempotence is enabled.
13:14:09/82 INFO  [main]: Kafka version: 2.4.0
13:14:09/82 INFO  [main]: Kafka commitId: 77a89fcf8d7fa018
13:14:09/82 INFO  [main]: Kafka startTimeMs: 1570264049533
Publishing record with value Wed Jan 02 13:14:09 AEDT 2020
13:14:09/495 INFO  [kafka-producer-network-thread | basic- ▯
    producer-sample]: [Producer clientId=basic-producer-sample] ▯
    Cluster ID: efkResGcSUWMV6zqj9D8vw
13:14:09/497 INFO  [kafka-producer-network-thread | basic- ▯
    producer-sample]: [Producer clientId=basic-producer-sample] ▯
    ProducerId set to 12000 with epoch 0
Published with metadata: getting-started-0@0, error: null
Publishing record with value Wed Jan 02 13:14:10 AEDT 2020
Published with metadata: getting-started-0@1, error: null
Publishing record with value Wed Jan 02 13:14:11 AEDT 2020
Published with metadata: getting-started-0@2, error: null
Publishing record with value Wed Jan 02 13:14:12 AEDT 2020
Published with metadata: getting-started-0@3, error: null
```

Consuming records

The following listing demonstrates how records are consumed.

```
import static java.lang.System.*;

import java.time.*;
import java.util.*;

import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.serialization.*;

public final class BasicConsumerSample {
  public static void main(String[] args) {
    final var topic = "getting-started";
```

```
    final Map<String, Object> config =
        Map.of(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9092",
               ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
               StringDeserializer.class.getName(),
               ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
               StringDeserializer.class.getName(),
               ConsumerConfig.GROUP_ID_CONFIG,
               "basic-consumer-sample",
               ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
               "earliest",
               ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
               false);

    try (var consumer = new KafkaConsumer<String, String>(config)) {
      consumer.subscribe(Set.of(topic));

      while (true) {
        final var records = consumer.poll(Duration.ofMillis(100));
        for (var record : records) {
          out.format("Got record with value %s%n", record.value());
        }
        consumer.commitAsync();
      }
    }
  }
}
```

This example is strikingly similar to the producer — same building of a config map, instantiation of a client, and the use of a *try-with-resources* block to ensure the client is closed after it leaves scope.

The first difference is in the configuration. Consumer have some elements common with producers — such as the `bootstrap.servers` list, and several others — but by and large they are different. The deserializer configuration is symmetric to the producer's serializer properties; there are key and value equivalents.

The group ID configuration is optional — it specifies the ID of the consumer group. This example uses a consumer group named `basic-consumer-sample`. The auto offset reset configuration stipulates what happens when the consumer subscribes to the topic for the first time. In this case, we would like the consumer's offset to be reset to the low-water mark for every affected partition, meaning that the consumer will get any backlogged records that existed prior to the creation of the group in Kafka. The default setting is `latest`, meaning the consumer will not read any prior records.

Finally, the auto-commit setting is disabled, meaning that the application will commit offsets at its discretion. The default setting is to enable auto-commit with a minimum interval of five seconds.

Prior to polling for records, the application must subscribe to one or more topics using the `Consumer.subscribe()` method.

Once subscribed, the application will repeatedly invoke `Consumer.poll()` in a loop, blocking up to a maximum specified duration or until a batch of records is received. For each received records, this example simply prints the record's value. Once all records have been printed, the offsets are committed asynchronously using the `Consumer.commitAsync()` method. The latter returns as soon the offsets are enqueued internally; the actual sending of the commit message to the group coordinator will take place on the background I/O thread. (The group coordinator is responsible for arbitrating the state of the consumer group.) The reader might also recall from Chapter 3: Architecture and Core Concepts, that the repeated polling and handling of records is called the *poll-process* loop.

Running the example above results in the following output (until terminated):

```
10:47:16/0     INFO  [main]: Kafka version: 2.4.0
10:47:16/0     INFO  [main]: Kafka commitId: 77a89fcf8d7fa018
10:47:16/0     INFO  [main]: Kafka startTimeMs: 1580341636585
10:47:16/2     INFO  [main]: [Consumer clientId=consumer-basic- ⏎
    consumer-sample-1, groupId=basic-consumer-sample] Subscribed ⏎
    to topic(s): getting-started
10:47:17/431   INFO  [main]: [Consumer clientId=consumer-basic- ⏎
    consumer-sample-1, groupId=basic-consumer-sample] Cluster ID: ⏎
    efkResGcSUWMV6zqj9D8vw
10:47:18/1740  INFO  [main]: [Consumer clientId=consumer-basic- ⏎
    consumer-sample-1, groupId=basic-consumer-sample] Discovered ⏎
    group coordinator 172.20.40.148:9092 (id: 2147483647 rack: null)
10:47:18/1744  INFO  [main]: [Consumer clientId=consumer-basic- ⏎
    consumer-sample-1, groupId=basic-consumer-sample] ⏎
    (Re-)joining group
10:47:18/1795  INFO  [main]: [Consumer clientId=consumer-basic- ⏎
    consumer-sample-1, groupId=basic-consumer-sample] ⏎
    (Re-)joining group
10:47:18/1828  INFO  [main]: [Consumer clientId=consumer-basic- ⏎
    consumer-sample-1, groupId=basic-consumer-sample] Finished ⏎
    assignment for group at generation 1: {consumer-basic-consumer- ⏎
    sample-1-26dce919-7f7d-4e04-98d9-99b091c73b3d=org.apache.kafka. ⏎
    clients.consumer.ConsumerPartitionAssignor$Assignment@66ea810}
10:47:18/1881  INFO  [main]: [Consumer clientId=consumer-basic- ⏎
    consumer-sample-1, groupId=basic-consumer-sample] Successfully ⏎
    joined group with generation 1
10:47:18/1884  INFO  [main]: [Consumer clientId=consumer-basic- ⏎
    consumer-sample-1, groupId=basic-consumer-sample] Adding newly ⏎
    assigned partitions: getting-started-1, getting-started-0, ⏎
```

```
    getting-started-2
10:47:18/1900  INFO  [main]: [Consumer clientId=consumer-basic- ⮠
    consumer-sample-1, groupId=basic-consumer-sample] Found no ⮠
    committed offset for partition getting-started-1
10:47:18/1900  INFO  [main]: [Consumer clientId=consumer-basic- ⮠
    consumer-sample-1, groupId=basic-consumer-sample] Found no ⮠
    committed offset for partition getting-started-0
10:47:18/1900  INFO  [main]: [Consumer clientId=consumer-basic- ⮠
    consumer-sample-1, groupId=basic-consumer-sample] Found no ⮠
    committed offset for partition getting-started-2
10:47:18/1921  INFO  [main]: [Consumer clientId=consumer-basic- ⮠
    consumer-sample-1, groupId=basic-consumer-sample] Resetting ⮠
    offset for partition getting-started-1 to offset 0.
10:47:18/1921  INFO  [main]: [Consumer clientId=consumer-basic- ⮠
    consumer-sample-1, groupId=basic-consumer-sample] Resetting ⮠
    offset for partition getting-started-0 to offset 0.
10:47:18/1921  INFO  [main]: [Consumer clientId=consumer-basic- ⮠
    consumer-sample-1, groupId=basic-consumer-sample] Resetting ⮠
    offset for partition getting-started-2 to offset 0.
Got record with value Wed Jan 02 13:14:09 AEDT 2020
Got record with value Wed Jan 02 13:14:10 AEDT 2020
Got record with value Wed Jan 02 13:14:11 AEDT 2020
Got record with value Wed Jan 02 13:14:12 AEDT 2020
Got record with value Wed Jan 02 13:14:13 AEDT 2020
Got record with value Wed Jan 02 13:14:14 AEDT 2020
Got record with value Wed Jan 02 13:14:15 AEDT 2020
Got record with value Wed Jan 02 13:14:16 AEDT 2020
Got record with value Wed Jan 02 13:14:17 AEDT 2020
```

This chapter has hopefully served as a practical reflection on the theoretical concepts that were outlined in Chapter 3: Architecture and Core Concepts. Specifically, we learned how to interact with a Kafka cluster using two distinct, yet commonly used approaches.

The part explored the use of the built-in CLI tools. These are basic utilities that allow a user to publish and consume records, administer topics and consumer groups, make various configuration changes, and query various aspects of the cluster state. As we have come to realise, the built-in tooling is far from perfect, but it is sufficient to carry out basic administrative operations, and at times it may be the only toolset at our disposal.

The second part looked at the programmatic interaction with Kafka, using the Java client library. We looked at simple examples for publishing and consuming records and learned the basics of the

Java client API. Real applications will undoubtedly be more complex than the provided examples, but they will invariably utilise the exact same building blocks.

Chapter 6: Design Considerations

Previous chapters have taken us through the essentials of event streaming and the core concepts of Kafka. By now, the reader should be familiar with the architecture of Kafka, its internal components, as well as the producer and consumer ecosystems. We have set up a Kafka broker, a Kafdrop UI and built basic producer and consumer applications using the Java client APIs.

In essence, the reader should now be equipped with the tools and foundational knowledge required to start building event streaming applications. But it takes time and experience to become proficient in Kafka. This is another way of saying: *To write good software, you need to make lots of mistakes.*

Mistakes need to be made; they are an essential part of learning. But learning from other peoples' mistakes is better than learning from one's own. So this chapter presents a list of considerations that are instructive in the design of performant and sustainable event streaming applications; considerations that have been amassed over years of working with these sorts of systems across a variety of industries.

Roles and responsibilities

Kafka permits a flexible arrangement between producers and consumers, allowing for a host of similar and disparate applications to interact with a topic simultaneously. In coming to terms with this, an often-asked question is: *Which party owns the topic, and who is responsible for its upkeep?*

Event-oriented broadcast

In a broadcast arrangement, where the producer-consumer relationship follows a (multi)point-to-multipoint topology, it is an accepted best-practice for the producer ecosystem to assume custodianship over the topic, and to effectively prescribe the entirety of the topic's configuration and usage semantics. These include —

- The lifecycle of the topic, as well as the associated broker-side configuration, such as the retention period and compaction policy;
- The nature and content of the published data, encodings, record schema, versioning strategy and associated deprecation period; and
- The sizing of the topic with respect to the partition count and the keying of the records.

In no uncertain terms: *the producer is king*. The producer will warrant all existential and behavioural aspects of the topic; the only decision left to the discretion of the consumer is whether to subscribe

to the topic or not. This, rather categorical, approach to role demarcation is essential to preserving the key characteristic of an event-driven architecture — *loose coupling*. The producer cannot be intrinsically aware of the topic's consumers, as doing so would largely defeat the intent of the design. This is not to say that producers should publish on a whim, or that the suitability of the published data is somehow immaterial to the outcome. Naturally, the published data should be correct, complete, and timely; however, the assurance of this lies with the designers of the system and is heavily predicated on the efficacy of domain modelling and stakeholder consultation. It is also evolutionary in nature; feedback from the consuming parties during the design, development, and operation phases should be used to iteratively improve the data quality. Stated otherwise, while the consuming parties are consulted as appropriate, the final decision rights and associated responsibilities rest with the producer.

Peer-to-peer messaging

Kafka may be used in peer-to-peer messaging arrangement, whereby the consumer is effectively responding to specific commands issued by a producer, and in most cases emitting responses back to the initiator. This model sees a role reversal: the consumer plays the role of the service provider, and therefore assumes custody over the lifecycle of the topic and its defining characteristics.

Where the response is sent over a different topic, shared among message initiators, the semantics of the response topic are also fully defined by the consumer. In more elaborate messaging scenarios, the initiator of the request may ask that the response is ferried over a dedicated topic to avoid sharing; in this case, the lifecycle of the topic and its retention will typically be managed by the initiator, while record-related aspects remain within the consumer's remit.

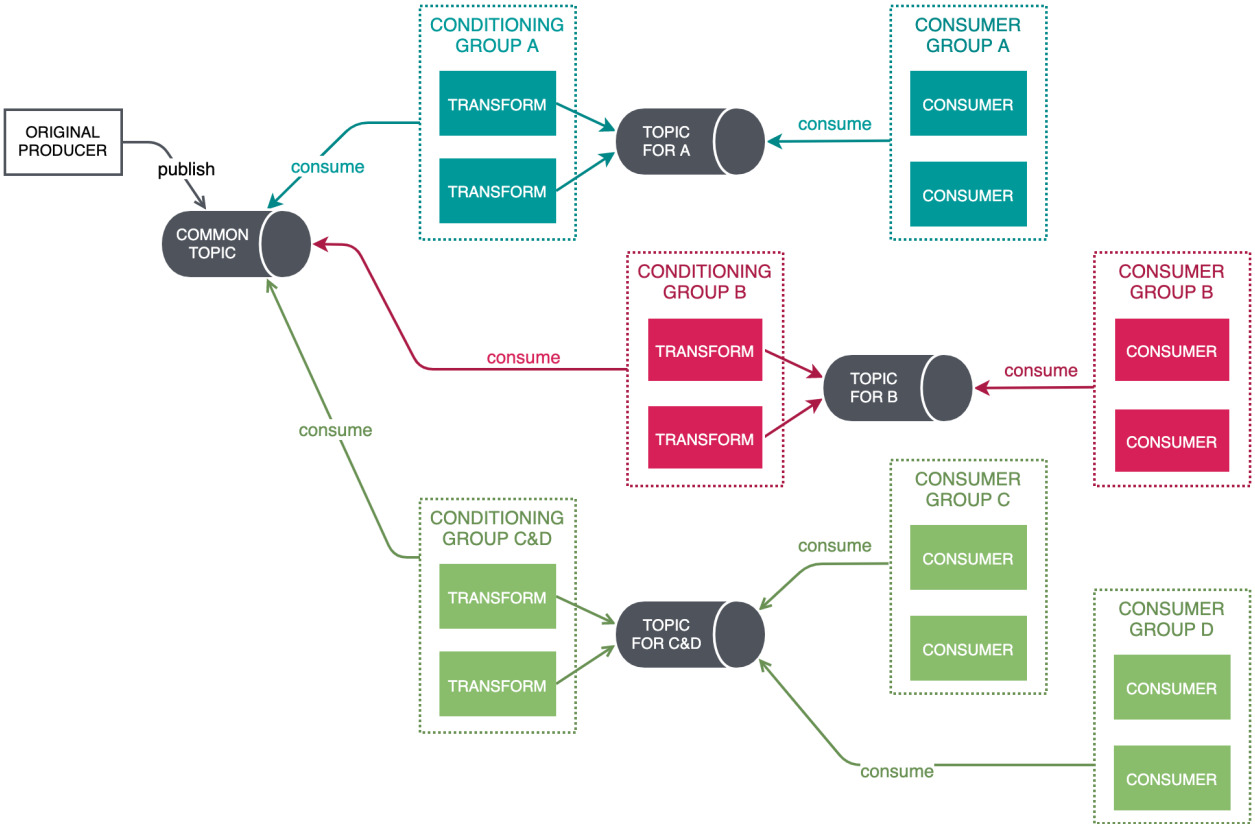
Topic conditioning

Reading the section on producer-driven topic modelling may fail to instill confidence in would-be consumers. The flip side of the coupling argument is the dreaded leap of faith. If the prerogative of the producer is to optimise topics around its domain model, what measures exist to assure the consumers that the upstream decisions do not impact them adversely? Is a compromise possible, and how does one neutralise the apparent bias without negatively impacting all parties.

These are fair questions. In answering, the reader is invited to consider the case where there are multiple disparate consumers. Truly, a single-producer-multiple-consumers is a fairly routine arrangement in contemporary event-driven architecture. And this is precisely the use case that highlights why a compromise is not a viable option. As the number of disagreeing parties grows, the likelihood of striking an effective compromise decreases to the point where the resulting solution is barely tractable for either party. It is the architectural equivalent of children fighting over a stuffed toy, where the inevitable outcome is the tearing of the toy, the dramatic scattering of its plush contents, resulting in discontent but ultimately quiesced children.

So what is one to do?

While this problem might not be trivially solvable, it can be readily compartmentalised. The use of a staged event-driven architecture (SEDA) offers a way of managing the complexity of diverse consumer requirements without negatively impacting the consumer applications directly or coupling the parties. Rather than feeding consumers directly off the producer-driven topic, intermediate processing stages are employed to condition the data to conform to an individual consumer group's expectations. These stages are replicated for each independent set of consumers, as shown in the diagram below.



Conditioning stages

Under this model, the impedance mismatch is resolved by the intermediate stages, leading to improved maintainability of the overall solution and allowing each of the end-parties to operate strictly within the confines of their respective domain models. The responsibilities of the parties are unchanged; the consumer takes ownership of the conditioning stage, responsible for its development and upkeep. While this might initially appear like a zero-sum transfer, the benefit of this approach is in its modularity. It embraces the single responsibility principle, does not clutter the consumer with transformational logic, and can lead to a more sustainable solution.

The acquired modularity may also lead to opportunities for component reuse. If two (or more) distinct consumer groups share similar data requirements, a common conditioning stage can power both. In the example above, a single conditioning stage powers both consumer groups *C* and *D*.

Parallelism

It was previously stated in Chapter 3: Architecture and Core Concepts that exploiting partial order enables consumer parallelism. The distribution of partition assignments among members of a consumer group is the very mechanism by which this is achieved. While consumer load-balancing is straightforward in theory, the practical implications of aspects such as topic sizing, record key selection, and consumer scaling are not so apparent.

There are several factors one must account for when designing highly performant event streaming applications. The following is an enumeration of some of these factors.

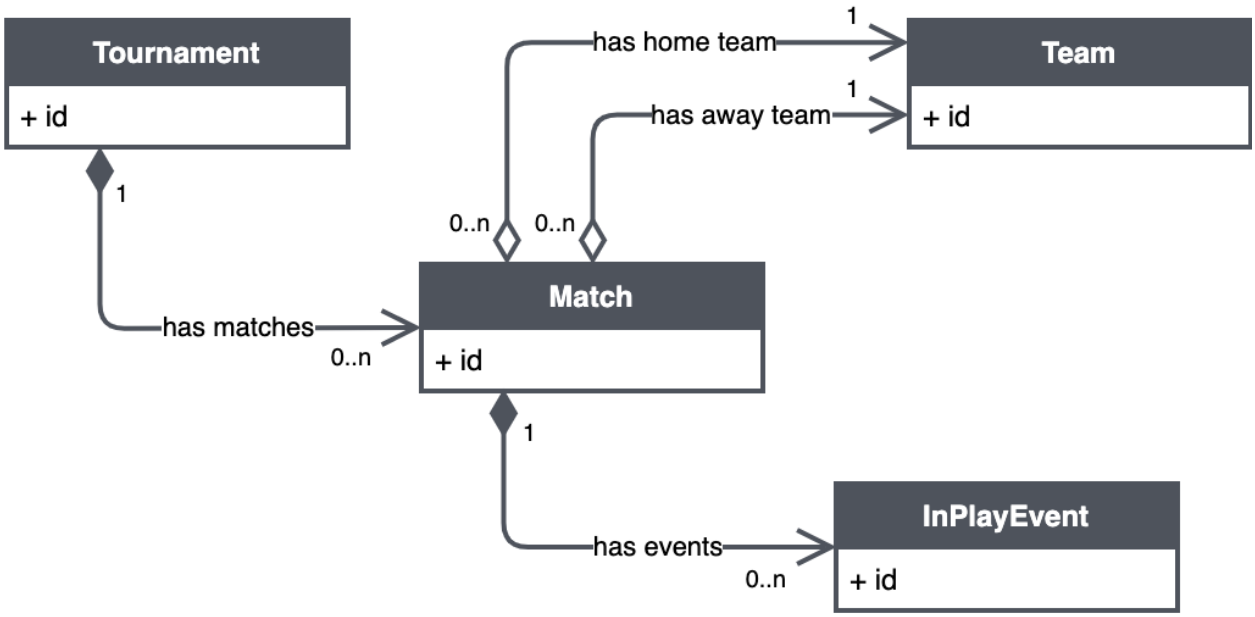
Producer-driven partitioning

Irrespective of the particular messaging topology employed, the responsibility of assigning records to partitions lies solely with the producer. This stems from a fundamental design limitation of Kafka; both topics and partitions are physical constructs, implemented as segmented log files under the hood. The publishing of a record results in the appending of a serialized form of the record to the head-end of an appropriate log file. Once this occurs, the relationship between a record and its encompassing topic-partition is cemented for the lifetime of the record.

In Kafka terminology, the most recent records are considered to be at the 'head' end of a partition.

The implication of this constraint is that the producer should take the utmost care in keying the records such as to preserve the essential causal relations, without overly constraining the record order. In practice, events will relate to some stable entity; the identifier of that entity can serve as the key of the corresponding record.

By way of example, consider a hypothetical content syndication system catering to football fans. Our system integrates with various real-time content providers, listening to significant in-play events from football matches as they unfold, then publishes a consolidated event stream to power multiple downstream consumers — mobile apps, scoreboards, player and match stats, video stream overlays, social networks such as Twitter, and other subscribers. True to the principles of event-driven architecture, we try to remain agnostic of what's downstream, focusing instead on the completeness and correctness of the emitted event stream — the data content of the records, their timing, and granularity. A simplified domain model for this contrived scenario is illustrated below.



Football domain model

Mimicking the real-life order of events is a good starting point for designing a streaming application. With this in mind, it makes sense to appoint the football match as the stable entity — its identifier will act as a record key to induce the partial order. This means that goals, corners, penalties, and so forth, will appear in the order they occurred within a match; records *across* matches will not be ordered by virtue of the matches being unrelated.

The basic requirement for the *stability* of the chosen entities is that they should persist for the lifetime of all causally related events. This does not imply that the entities should be long-lived, only that they exist long enough to survive the causal chains that depend on them. To our earlier example, the football match survives all of its in-play events.

Record keys bound to some stable identity can be thought of as the equivalent of *foreign keys* in database parlance, and the stability characteristic can be likened to a *referential integrity* constraint. Exploring the relational database analogy, the entity in the linked table would persist for as long as it is being referenced by one or more foreign keys. Likewise, the linked stable entity persists for the duration of time from the point when the first record is published to when the last record leaves the producer. Thereafter, once the causal order is materialised in Kafka, the linked entity becomes largely irrelevant.

Assuming no special predecessor-successor relationship between the chosen stable entities, the partial order of the resulting records will generally be sufficient for most, if not all, downstream consumers. Conversely, if the entities themselves exhibit causal relationships, then the resulting stream may fail to capture the complete causality. Reverting to our earlier example, suppose one downstream subscriber is a tournament leaderboard application, needing to capture and present the relative standing of the teams as they progress through a tournament. Match-centric order may be insufficient, as goals would need to be collated in tournament order.

Where a predecessor-successor relationship exists between stable entities, and that relationship is significant to downstream subscribers, there are generally two approaches one may take. The first is to coarsen the granularity of event ordering, for example, using the tournament identifier is the record key. This preserves the chronological order of in-play events within a tournament, and by extension, within a match. The second approach is to transfer the responsibility of event reordering to the downstream subscriber.

Coarsening of causal chains is generally preferred when said order is an intrinsic characteristic of the publisher's domain. In our example, the publisher is well aware of the relationship between tournaments and matches, so why not depict this?

The main drawback of this approach is the reduced opportunity for consumer parallelism, as coarsening leads to a reduction in the cardinality of the partitioning key. Subscribers will not be able to utilise Kafka's load-balancing capabilities to process match-level substreams in parallel; only tournaments will be subject to parallelism. This may satisfy some subscribers, but penalise others.

An extension of this model is to introduce conditioning stages for those subscribers that would benefit from the finer granularity. A conditioning stage would consume a record from a coarse-grained input topic, then republish the same value to an output topic with a different key — for example, switching the key from a tournament ID to a match ID. This model nicely dovetails into the broader principle of producer-driven domain modelling, while satisfying individual subscriber needs with the help of SEDA.

The second approach — fine-grained causal chains with consumer-side reordering — assumes that the consumer, or some intermediate stage acting on its behalf, is responsible for coarsening the granularity of the event substreams to fit some bespoke processing need. The conditioning stage will need to be stateful, maintaining a staging datastore for incoming events so that they can be reordered. In practice, this is much more difficult than it appears. In some cases, there simply isn't enough data available for a downstream stage to reconstruct the original event order. In short, consumer-side reordering may not always a viable option.

There is no one-size-fits-all approach to partitioning domain events. As a rule of thumb, the producer should publish at the finest level of granularity that makes sense in its respective domain, while supporting a diverse subscriber base. A crucial point: being agnostic of subscribers does not equate to being ignorant of their needs. The effective modelling of the domain and the associated event streams fall on the shoulders of architects and senior technologists; stakeholder consultation and understanding of the overall landscape are essential in constructing a sustainable solution.

In the absence of consumer awareness, one litmus test for formulating partial order is to ascertain that the resulting stream can be used to reconstitute the source domain. Ask the question: Can a hypothetical subscriber rebuild an identical replica of the domain purely from the emitted events while maintaining causal consistency? If the answer is 'yes', then the event stream should be suitable for any downstream subscriber. Otherwise, if the answer is 'no', then there is a gap in the condition of the emitted events that requires attention.

Topic width

With the correct keying granularity in place, the next consideration is the 'width' of the topic — the number of partitions it encompasses. Assuming a fair distribution of keys, increasing the number of partitions creates more opportunities for the consumer ecosystem to process records in parallel. Getting the topic width right is essential to a performant event streaming architecture.

Regrettably, Kafka does not make this easy for us. Kafka only permits non-destructive resizing of topics when *increasing* the partition count. Decreasing the number of partitions is a destructive operation — requiring the topic to be created anew, and manually repopulated.

One of the main gotchas of resizing topics is the effect they have on record order. As stated in Chapter 3: Architecture and Core Concepts, a Kafka producer hashes the record's key to arrive at the partition number. The hashing scheme is not *consistent* — two records with the same key hashed at different points in time will correspond to an identical partition number *if and only if* the number of partitions has not changed in that time. Increasing the partition count results in the two records occupying potentially different partitions — a clear breach of Kafka's key-centric ordering guarantee. Kafka will not rehash records as part of resizing, as this would be prohibitively expensive in the absence of consistent hashing. (To be clear, consistent hashing would not eliminate the need for rehashing records *per se*, but it would dramatically reduce the number of affected hashes when the topic is widened.)

When the correctness of a system is predicated on the key-centric ordering of records, avoid resizing the topic as this will effectively void any ordering guarantees that the consumer ecosystem may have come to rely upon.

One approach to dealing with prospective growth is to start with a sufficiently over-provisioned topic, perhaps an order of magnitude more partitions than one would reasonably expect — thereby avoiding the hashing skew problem down the track. On the flip side, increasing the number of partitions may increase the load on the brokers and consumers.

A partition is backed by log files, which require additional file descriptors. (At minimum, there is one log segment and one index file per partition.) Inbound writes flow to dedicated buffers, which are allocated per partition on the broker. Therefore, increasing the number of partitions, in addition to consuming extra file handles, will result in increased memory utilisation on the brokers. A similar impact will be felt on consumers, sans the file handles, which also employ per-partition buffers for fetching records.

A further impact of wide topics may be felt due to the limitations of the inter-broker replication process and its underlying threading model. Specifically, a broker will allocate one thread for every other broker that it maintains a connection with, which covers the set of replicated partitions — where the two peers have a leader-follower relationship. The replication threads may act as a bottleneck when shuttling records from the partition leader to in-sync replicas, and thereby impact the publishing latency when the producer requests all replicas to acknowledge the writes. This problem is ameliorated when the number of brokers increases, as the growth of the cluster has

a downward effect on the ratio of partitions to brokers, taking the pressure off each connection. Confluent — one of the major contributors to Apache Kafka — recommends limiting the number of partitions per broker to $100 \times b \times r$, where b is the number of brokers in a Kafka cluster and r is the replication factor.

So while a wider topic provides for greater theoretical throughput, it does carry practical and immediate implications on the client and broker performance. The impacts of widening individual topics may not be substantial, but they may be felt in aggregate. This is not to say that topics should not be over-provisioned; rather, decisions regarding the sizing of the topics and the extent of over-provisioning should not be taken on a whim. The broker topology and the overall capacity of the cluster play a crucial role; these have to be adequately specified and taken into consideration when sizing the topics.

If dealing with an existing topic that has saturated its capacity for consumer parallelism, consider a staged destructive resize. Create a new topic of the desired size, using a tool such as *MirrorMaker* to replicate the topic contents onto the wider topic. When the replication catches up, switch the producers to double-publish to both the old and the new topics. There may need to be some producer downtime to allow for topic parity. Individual consumer groups can start migrating to the new topic at their discretion; however, the misalignment of partitions may present a challenge with persisted offsets. Assuming that consumers have been designed with idempotency in mind, one should be able to set the `auto.offset.reset` property to `earliest` to force the reprocessing of the records from the beginning. Depending on the retention of the topic, this may take some time, which will also delay the consumers' ability to process new records. Alternatively, one can reset the offsets to a specific timestamp, which will substantially reduce the quantity of replayed records. Some reprocessing will likely be unavoidable; such is the price for resizing strongly-ordered topics. (The double-publishing code can be removed when all consumers have been migrated.)

Scaling of the consumer group

Kafka will allocate partitions approximately evenly among members of a consumer group, up to the width of the topic. So to increase parallelism, one must ensure sufficient consumer instances in the group. Allocating a fixed number of instances to the group is usually not economical, as it may result in idle capacity, particularly for event streams that exhibit cyclic or bursty loading. The recommended approach is to employ an automated horizontal scaling technique to dynamically expand or contract the population of the group in response to load demand. For example, if deploying the consumer group within a public cloud environment like AWS, one may use an autoscaling group to provision additional instances based on CPU utilisation metrics. Alternatively, if the consumer application is containerised, the use of a container orchestration platform is recommended. For example, when deployed in Kubernetes, one would employ horizontal pod autoscaling to dynamically size the consumer group.

Internal consumer parallelism

An alternate way of increasing consumer throughput, without widening the topic or scaling the consumer group, is to exploit parallelism within the consumer process. This can be achieved by partitioning the workload among a pool of threads by independently hashing the record keys to maintain local order. This strategy would be classed as vertical scaling, requiring increased parallelism on each consumer node in exchange for reducing the number of consumers, and hence the number of partitions.

Idempotence and exactly-once delivery

Chapter 3: Architecture and Core Concepts had introduced the concepts of delivery guarantees, stating that Kafka allows for two different modes of delivery by simply shifting the point when the consumer commits its offsets. *At-least once* and *at-most-once* guarantees were named, but there was no mention of an *exactly-once* guarantee. This might be a good segue to discuss the differences between delivery modes; ultimately, it will help us understand what it means to do something 'exactly once'.

The role of messaging middleware is to decouple communications between collaborating parties. When a sender publishes a message, there is an assumption that the receiver (or receivers, as there may be multiple such parties) will eventually consume and process the message. Messaging middleware is generally divided into two categories: those that offer *at-most-once* and those that offer *at-least-once* guarantees. And then we have Kafka, which has a foot in each camp.

The at-most-once guarantee simply means that a message is never redelivered to its recipient, no matter the contingency. The consumer might read the record, but then fail for whatever reason before processing the record. If the offsets for the said record were committed before the record was processed, then the reassignment of the partition following the consumer's failure will result in the skipping of the record by the new consumer.

Some messaging systems are truly *fire-and-forget*, in the sense that there might not even be an initial attempt to deliver the message in the first place. For example, some message brokers support load shedding, in which case a message might be purged from the queue to avoid accumulating a backlog of stale messages. Other messaging systems, such as ZeroMQ, might be purely in-memory, in which case the loss of a node will result in the loss of undelivered messages. Some systems use the term 'maybe-once' as a stand-in for 'at-most-once', which seems more fitting in some cases. Kafka's take on at-most-once processing is slightly different from some of its counterparts. Provided that a record has been stably persisted to a topic, and is within the retention period, Kafka will always allow the consumer to read the record at least once, no matter what. So the term 'at-most-once' applies to the processing of the record, rather than to the mere act of reading the record.

The at-least-once guarantee means that a message will only be marked as delivered when it completes its entire journey within the consumer application. Failure prior to this point is treated as non-delivery and a retry will ensue.

The term 'delivery' may seem somewhat confusing, especially if one perceives delivery in a postal sense. Delivery is not just leaving a record at the consumer's doorstep, but seeing the consumer 'sign' for the delivery by committing the record's offset.

Using the at-most-once approach for delivery is acceptable in many cases, especially where the occasional loss of a record does not leave a system in a perpetually inconsistent state. At-most-once delivery is useful where the source of the record is continuously, within a fixed interval, emitting updates to some entity of interest, such that the loss of one record can be recovered from in bounded time. Conversely, the at-least-once approach is more fitting where the loss of a record constitutes an irreversible loss of data, violating some fundamental invariant of the system. But the flip side is that processing a record multiple times may introduce undesirable side-effects. This is where the notion of *exactly-once* processing enters the scene. In fact, when contrasting at-least-once with at-most-once delivery semantics, an often-asked question is: *Why can't we just have it once?*

Without delving into the academic details, which involve conjectures and impossibility proofs, it is sufficient to say that exactly-once semantics are not possible without tight-knit collaboration with the consumer application. As disappointing as it may sound, a messaging platform cannot offer exactly-once guarantees on its own. What does this mean in practice?

To achieve the coveted *exactly-once* semantics, consumers in event streaming applications must be *idempotent*. In other words, processing the same record repeatedly should have no net effect on the consumer ecosystem. If a record has no additive effects, the consumer is inherently idempotent. (For example, if the consumer simply overwrites an existing database entry with a new one, then the update is naturally idempotent.) Otherwise, the consumer must check whether a record has already been processed, and to what extent, prior to processing the record. *The combination of at-least-once delivery and consumer idempotence collectively leads to exactly-once semantics.*

The design of an idempotent consumer mandates that all effects of processing a record must be traceable back to the record. For example, a record might require updating a database, invoking some service API, or publishing one or more records to a set of downstream topics. The latter is particularly common in SEDA systems, which are essentially graphs of processing nodes joined by topics. When a consumer processes a record, it will have no awareness of whether the record is being processed for the first time, or whether the given record is a repeat attempt of an earlier failed delivery. As such, *the consumer must always assume that a record is a duplicate*, and handle it accordingly. Every potential side-effect must be checked to ensure that it hasn't already occurred, before attempting it a second time. When a side-effect is itself idempotent, then it can be repeated unconditionally.

In some cases, there may not be an easy way to determine whether a potential side-effect had already occurred as a result of a previous action. For example, the side-effect might be to publish a record on another topic; there is often no practical way of querying for the presence of a prior record. Kafka offers an advanced mechanism for correlating the records consumed from input topics with the resulting records on output topics — this is discussed in Chapter 18: Transactions. Transactions can create joint atomicity and isolation around the consumption and production of records, such that either all scoped actions appear to have occurred, or none.

Where the target endpoint is a (non-Kafka) message queue, the downstream receiver must be made idempotent. In other words, two (or more) identical records with different offsets must not result in material duplication somewhere down the track. This is called *end-to-end idempotence*. As the name suggests, this guarantee spans the entirety of an event-streaming graph, covering all nodes and edges. In practice, this is achieved by ensuring that any two neighbouring nodes have an established mechanism for idempotent communication.

This chapter has explored some of the fundamental considerations pertinent to the design and construction of safe and performant event streaming applications.

We started by covering the roles and responsibilities of the various parties collaborating in the construction of distributed event-driven applications. The key takeaway is that the parties publishing or consuming events can be likened to service providers and invokers, and their roles vary depending on the messaging topology. We also explored scenarios where producers and consumers might disagree on the domain model, and the methods by which this can be resolved.

The concept of key-centric record parallelism — Kafka's trademark performance enhancer — has been explored. We looked at the factors that constrain the consumers' ability to process events in parallel, and the design considerations that impact the producing party.

Finally, we contrasted at-most-once and at-least-once delivery guarantees and arrived at the design requirements for exactly-once — namely, a combination of at-least-once delivery and consumer idempotence.

Chapter 7: Serialization

The examples we have come across so far demonstrated fundamental Kafka producer and consumer behaviour by serializing basic types, such as strings. While this may be sufficient to garner an introductory level of awareness, it is of limited use in practice, as real-life applications rarely publish or consume unstructured strings.

Distributed applications communicating in either a message-passing style or as part of an event-driven architecture will utilise a broad catalogue of structured datatypes and corresponding schema contracts. The business logic embedded in producer and consumer applications will typically deal with native domain models, requiring a bridging mechanism to marshal these models to Kafka topics when producing records, and perform the opposite when consuming from a topic.

This chapter covers the broad topic of record serialization. In the course of the discussion, we shall also explore complementary design patterns that streamline the interfacing of an application's business logic with the underlying event stream.

Key and value serializer

Prior examples have revealed that the Kafka `Producer` and `Consumer` API, as well as the `ProducerRecord` and `ConsumerRecord` classes, are generically typed. The `Producer` interface is parametrised with a key and a value type, denoted `K` and `V` in the type parameter list:

```
/**
 * The interface for the {@link KafkaProducer}
 * @see KafkaProducer
 * @see MockProducer
 */
public interface Producer<K, V> extends Closeable {
  /** some methods omitted for brevity */

  /**
   * See {@link KafkaProducer#send(ProducerRecord)}
   */
  Future<RecordMetadata> send(ProducerRecord<K, V> record);

  /**
   * See {@link KafkaProducer#send(ProducerRecord, Callback)}
   */
```

```
  Future<RecordMetadata> send(ProducerRecord<K, V> record,
                              Callback callback);
}
```

The `send()` methods reference the type parameters, requiring the supplied `ProducerRecord` to be of a matching generic type.

Kafka's ingrained type-safety mechanism assumes a pair of compatible serializers for supported key and value types. A custom serializer must conform to the `org.apache.kafka.common.serialization.Serializer` interface, listed below.

```
public interface Serializer<T> extends Closeable {
  default void configure(Map<String, ?> configs, boolean isKey) {
    // intentionally left blank
  }

  byte[] serialize(String topic, T data);

  default byte[] serialize(String topic, Headers headers, T data) {
    return serialize(topic, data);
  }

  @Override
  default void close() {
    // intentionally left blank
  }
}
```

Serializers are configured in one of two ways:

1. Passing the fully-qualified class name of a `Serializer` implementation to the producer via the `key.serializer` and the `value.serializer` properties. Note, there are no default values assigned to these properties.
2. Directly instantiating the serializer and passing it as a reference to an overloaded `KafkaProducer` constructor.

The property-based mechanism has the advantage of simplicity, in that it lives alongside the rest of the producer configuration. One can look at the configuration properties and instantly determine that the producer is configured with a specific key and value serializer. The drawback of this configuration style is that it requires the `Serializer` implementation to include a public, no-argument constructor. It also makes it difficult to configure. Because the serializer is instantiated reflectively by the producer client, the application code is unable to inject its own set of arguments at the point of initialisation. The only way to configure a reflectively-instantiated serializer is to supply a set of custom properties to the producer, then retrieve the values of these properties in the `Serializer` implementation, using the optional `configure()` callback:

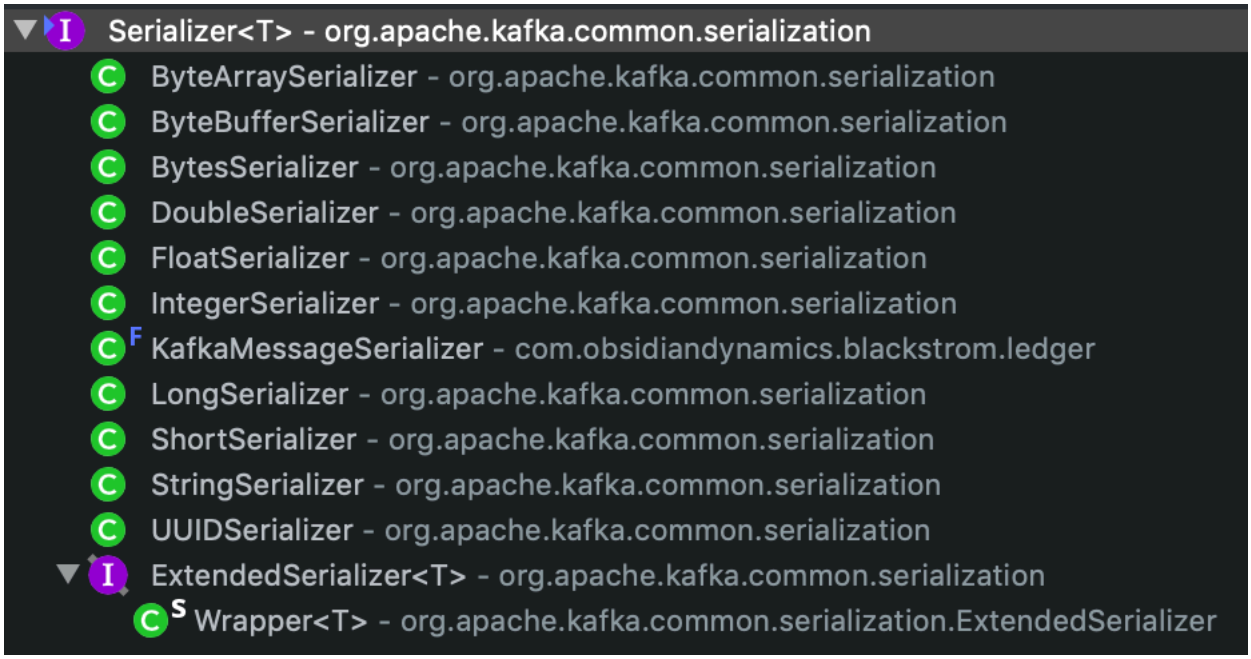
```
/**
 * Configure this class.
 * @param configs configs in key/value pairs
 * @param isKey whether is for key or value
 */
default void configure(Map<String, ?> configs, boolean isKey) {
  // intentionally left blank
}
```

The `configure()` method also helps the instance determine whether it is a key or a value serializer.

Another drawback of the property-based approach is that it ignores the generic type constraints imposed by the `Producer` interface. For example, it is possible to instantiate a `KafkaProducer<Integer, String>` using a `FloatSerializer` for the key and a `ByteArraySerializer` for the value. The problem will remain unnoticed until you try to publish the first record, which will summarily fail with a `ClassCastException`.

Compared to the property-based approach, the passing of a pre-instantiated serializer to a `KafkaProducer` simultaneously solves the problems of maintaining generic type-safety and the configuration of the `Serializer` instance. The application code will instantiate a `Serializer` and configure it appropriately before invoking the `KafkaProducer` constructor. In turn, the constructor's signature will ensure that the given pair of `Serializer` instances conform to the `K` and `V` generic type parameters.

The Kafka client library comes with several pre-canned serializers for common data types, listed below.



Supported serializers

In most applications, record *keys* are simple unstructured values such as integers, strings, or UUIDs, and a built-in serializer will suffice. Record *values* tend to be structured payloads conforming to some pre-agreed schema, represented using a text or binary encoding. Typical examples include JSON, XML, Avro, Thrift, and Protocol Buffers. When serializing a custom payload to a Kafka record, there are generally two approaches one may pursue. These are:

1. Implement a custom serializer to directly handle the payload.
2. Serialize the payload at the application level.

The first approach is idiomatic; unquestionably, it is more fitting to the design of the Kafka API. We would subclass `Serializer`, implementing its `serialize()` method:

```
/**
 * Convert {@code data} into a byte array.
 *
 * @param topic topic associated with data
 * @param data typed data
 * @return serialized bytes
 */
byte[] serialize(String topic, T data);
```

The `serialize()` method accepts the `data` argument that is typed in accordance with the generic type constraint of the `Serializer` interface. From there it is just a matter of marshalling the payload to a byte array.

The alternate method involves piggybacking on an existing serializer that matches the underlying encoding. When dealing with text-based formats, such as JSON or XML, one would use the `StringSerializer`. Conversely, when dealing with binary data, the `ByteArraySerializer` would be selected. This leaves Kafka's `ProducerRecord` and `Producer` instance unaware of the application-level datatype, relying on the application code to pre-serialize the value before constructing a `ProducerRecord`.

A potential advantage of a custom Kafka serializer over application-level serialization is the additional type-safety that the former offers. Looking at it from a different lens, the need for type-safety at the level of a Kafka producer is questionable, as it would likely be encapsulated within a dedicated messaging layer.

This is a good segue into layering. A well-thought-out application will clearly separate business logic from the persistence and messaging concerns. For example, you don't expect to find JDBC `Connection` instances scattered unceremoniously among the business logic classes of a well-designed application. (Nor are JDBC classes type-safe for that matter.) By the same token, it makes sense for the `Producer` class to also be encapsulated in its own layer, ideally using an interface that allows the messaging code to be mocked out independently as part of unit testing. Throughout the chapter,

we will list arguments in favour of encapsulating common messaging concerns within a dedicated layer.

Returning to the question of a custom serializer versus a piggybacked approach, the former is the idiomatic approach, and for this reason we will stick with custom (de)serializers throughout the chapter.

Sending events

For the forthcoming discussion, consider a contrived event streaming scenario involving a basic application for managing customer records.

The complete source code for the upcoming examples is available at github.com/ekoutanov/effectivekafka[8] in the `src/main/java/effectivekafka/customerevents` directory. Most of the relevant code listings are also included here for the reader's convenience; some of the more esoteric items may have been omitted, but they should nonetheless be present in the `effectivekafka` repository on GitHub.

Every change to the customer entity results in the publishing of a corresponding event to a single Kafka topic, keyed by the customer ID. Each event is strongly typed, but there are several event classes and each is bound to a dedicated schema. The POJO representation of these events might be `CreateCustomer`, `UpdateCustomer`, `SuspendCustomer`, and `ReinstateCustomer`. The abstract base class for all customer-related events will be `CustomerPayload`. The base class also houses the common fields, which for the sake of simplicity have been reduced to a single UUID-based unique identifier. This is the ID of the notional customer entity to which the event refers. (For simplicity, the examples will not contain the persistent entities — just the event notifications.)

We are going to assume that records should be serialized using JSON. Along with Avro, JSON is one of the most popular formats for streaming event data over Kafka. The examples in this book use the *FasterXML Jackson* library for working with JSON, which is the *de facto* JSON parser within the Java ecosystem. Subclasses of `CustomerPayload` are specified using a `@JsonSubTypes` annotation, which allows us to use Jackson's built-in support for polymorphic types. Every serialized `CustomerPayload` instance will contain a `type` property, specifying an aliased name of the concrete type, for example, `CREATE_CUSTOMER` for the `CreateCustomer` class. Jackson uses this property as a hint during deserialization, picking the correct subclass of `CustomerPayload` to map the JSON document to.

[8] https://github.com/ekoutanov/effectivekafka/tree/master/src/main/java/effectivekafka/customerevents

```
import java.util.*;

import com.fasterxml.jackson.annotation.*;

@JsonTypeInfo(use=JsonTypeInfo.Id.NAME,
              include=JsonTypeInfo.As.EXISTING_PROPERTY,
              property="type")
@JsonSubTypes({
  @JsonSubTypes.Type(value=CreateCustomer.class,
                     name=CreateCustomer.TYPE),
  @JsonSubTypes.Type(value=UpdateCustomer.class,
                     name=UpdateCustomer.TYPE),
  @JsonSubTypes.Type(value=SuspendCustomer.class,
                     name=SuspendCustomer.TYPE),
  @JsonSubTypes.Type(value=ReinstateCustomer.class,
                     name=ReinstateCustomer.TYPE)
})
public abstract class CustomerPayload {
  @JsonProperty
  private final UUID id;

  CustomerPayload(UUID id) {
    this.id = id;
  }

  public abstract String getType();

  public final UUID getId() {
    return id;
  }

  protected final String baseToString() {
    return "id=" + id;
  }
}
```

Exposing a stable alias rather than the fully-qualified Java class name makes our message schema portable, enabling us to interoperate with non-Java clients. This also aligns with the cornerstone principle of event-driven architecture — the producer has minimal awareness of the downstream consumers, and makes no assumption as to their role and cause, nor their implementation.

For a sampling of a typical concrete event, we have the `CreateCustomer` class. There are a few others,

but they are conceptually similar.

```
public final class CreateCustomer extends CustomerPayload {
  static final String TYPE = "CREATE_CUSTOMER";

  @JsonProperty
  private final String firstName;

  @JsonProperty
  private final String lastName;

  public CreateCustomer(@JsonProperty("id") UUID id,
                        @JsonProperty("firstName") String firstName,
                        @JsonProperty("lastName") String lastName) {
    super(id);
    this.firstName = firstName;
    this.lastName = lastName;
  }

  @Override
  public String getType() {
    return TYPE;
  }

  public String getFirstName() {
    return firstName;
  }

  public String getLastName() {
    return lastName;
  }

  @Override
  public String toString() {
    return CreateCustomer.class.getSimpleName() + " ["
        + baseToString() + ", firstName=" + firstName
        + ", lastName=" + lastName + "]";
  }
}
```

Ideally, we would like to inject a high-level event sender into the business logic, then have our business logic invoke the sender whenever it needs to produce an event, without concerning itself with how the event is serialized or published to Kafka. This is the perfect case for an interface:

```
import java.io.*;
import java.util.concurrent.*;

import org.apache.kafka.clients.producer.*;

public interface EventSender extends Closeable {
  Future<RecordMetadata> send(CustomerPayload payload);

  final class SendException extends Exception {
    private static final long serialVersionUID = 1L;

    SendException(Throwable cause) { super(cause); }
  }

  default RecordMetadata blockingSend(CustomerPayload payload)
      throws SendException, InterruptedException {
    try {
      return send(payload).get();
    } catch (ExecutionException e) {
      throw new SendException(e.getCause());
    }
  }

  @Override
  public void close();
}
```

The application might want to send records asynchronously — continuing without waiting for an outcome, or synchronously — blocking until the record has been published. We have specified a `Future<RecordMetadata> send(CustomerPayload payload)` method signature for the asynchronous operation — to be implemented by the concrete `EventSender` subclass. The synchronous case is taken care of by the `blockingSend()` method, which simply delegates to `send()`, blocking on the result of the returned `Future`. The sender implementation may choose to override this method with a more suitable one if need be. (Hopefully, the default implementation is good enough.)

Next, we are going to look at a sample user of `EventSender` — the `ProducerBusinessLogic` class.

```
public final class ProducerBusinessLogic {
  private final EventSender sender;

  public ProducerBusinessLogic(EventSender sender) {
    this.sender = sender;
  }

  public void generateRandomEvents()
      throws SendException, InterruptedException {
    final var create =
        new CreateCustomer(UUID.randomUUID(), "Bob", "Brown");
    blockingSend(create);

    if (Math.random() > 0.5) {
      final var update =
          new UpdateCustomer(create.getId(), "Charlie", "Brown");
      blockingSend(update);
    }

    if (Math.random() > 0.5) {
      final var suspend = new SuspendCustomer(create.getId());
      blockingSend(suspend);

      if (Math.random() > 0.5) {
        final var reinstate = new ReinstateCustomer(create.getId());
        blockingSend(reinstate);
      }
    }
  }

  private void blockingSend(CustomerPayload payload)
      throws SendException, InterruptedException {
    System.out.format("Publishing %s%n", payload);
    sender.blockingSend(payload);
  }
}
```

We are not actually going to implement any life-like business logic for this example; the intention is merely to simulate some activity and exercise our future `EventSender` implementation.

The complete sender

Using interfaces is only going to get us so far; we need a concrete implementation of a `EventSender` to make the example work. Here is the simplest sender implementation that will get the job done:

```
import java.util.*;
import java.util.concurrent.*;

import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.*;

public final class DirectSender implements EventSender {
  private final Producer<String, CustomerPayload> producer;

  private final String topic;

  public DirectSender(Map<String, Object> producerConfig,
                      String topic) {
    this.topic = topic;

    final var mergedConfig = new HashMap<String, Object>();
    mergedConfig.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                     StringSerializer.class.getName());
    mergedConfig.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                     CustomerPayloadSerializer.class.getName());
    mergedConfig.putAll(producerConfig);
    producer = new KafkaProducer<>(mergedConfig);
  }

  @Override
  public Future<RecordMetadata> send(CustomerPayload payload) {
    final var record =
        new ProducerRecord<>(topic,
                             payload.getId().toString(),
                             payload);
    return producer.send(record);
  }

  @Override
  public void close() {
    producer.close();
  }
}
```

There really isn't much to it. The `DirectSender` encapsulates a `KafkaProducer`, configured using a supplied map of properties. The constructor will overwrite certain key properties in the user-specified configuration map — properties that are required for the correct operation of the sender and should not be interfered with by external code. The `DirectSender` also requires the name of the topic.

The `send()` method simply creates a new `ProducerRecord` and enqueues it for sending, delegating to the underlying `Producer` instance. Before this, the `send()` method will also assign the newly created record's key, which, as previously agreed, is expected of the producer application. By setting the key to the customer ID, we ensure that records are strictly ordered by customer.

The benefits of layering become immediately apparent. Without an `EventSender` implementation to guide the construction and sending of records, the responsibility of enforcing invariants would have rested with the business logic layer. This prevents us from enforcing simple invariants that operate at record scope, such as "the key of a record must equal to the ID of the encompassed customer event". Relying on the business logic to set the key correctly is error-prone, especially when you consider that there will be several places where this would be done. By layering our producer application, we can enforce this behaviour deeper in the stack, thereby minimising code duplication and avoiding a whole class of potential bugs.

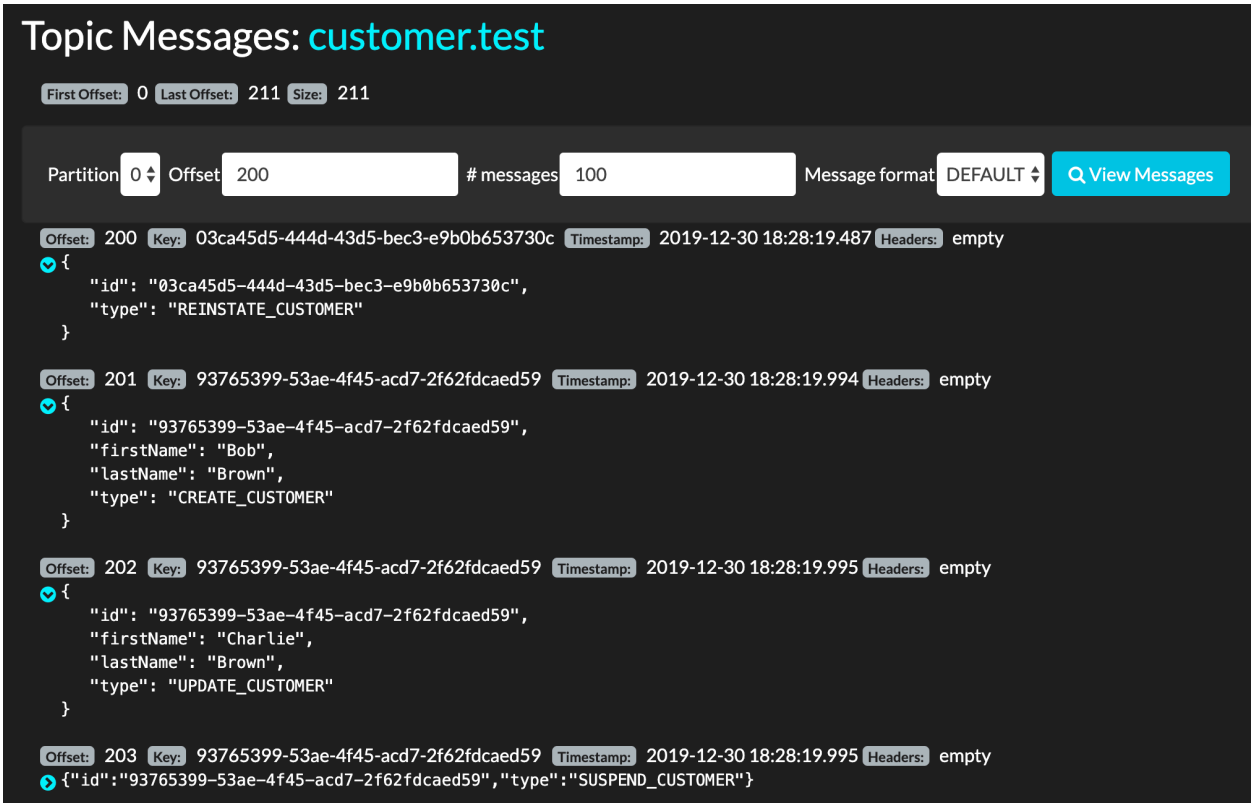
To try out the example, first launch the `RunRandomEventProducer` class. As the name suggests, it will publish a sequence of random customer events using the `ProducerBusinessLogic` defined earlier.

```
public final class RunRandomEventProducer {
  public static void main(String[] args)
      throws InterruptedException, SendException {
    final Map<String, Object> config =
        Map.of(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9092",
               ProducerConfig.CLIENT_ID_CONFIG,
               "customer-producer-sample");

    final var topic = "customer.test";

    try (var sender = new DirectSender(config, topic)) {
      final var businessLogic = new ProducerBusinessLogic(sender);
      while (true) {
        businessLogic.generateRandomEvents();
        Thread.sleep(500);
      }
    }
  }
}
```

Assuming everything goes well, you should see some logs printed to standard out:

```
...
omitted for brevity
...
Publishing {"id":"28361a15-7cef-47f3-9819-f8a629491c5a", □
    "firstName":"Bob","lastName":"Brown","type":"CREATE_CUSTOMER"}
Publishing {"id":"28361a15-7cef-47f3-9819-f8a629491c5a", □
    "type":"SUSPEND_CUSTOMER"}
Publishing {"id":"28361a15-7cef-47f3-9819-f8a629491c5a", □
    "type":"REINSTATE_CUSTOMER"}
Publishing {"id":"b3cd538c-90cc-4f5f-a5dc-b1c469fc0bf8", □
    "firstName":"Bob","lastName":"Brown","type":"CREATE_CUSTOMER"}
Publishing {"id":"b3cd538c-90cc-4f5f-a5dc-b1c469fc0bf8", □
    "firstName":"Charlie","lastName":"Brown","type":"UPDATE_CUSTOMER"}
Publishing {"id":"83bb14e7-9139-4699-a843-8b3a90ae26e2", □
    "firstName":"Bob","lastName":"Brown","type":"CREATE_CUSTOMER"}
Publishing {"id":"83bb14e7-9139-4699-a843-8b3a90ae26e2", □
    "firstName":"Charlie","lastName":"Brown","type":"UPDATE_CUSTOMER"}
Publishing {"id":"83bb14e7-9139-4699-a843-8b3a90ae26e2", □
    "type":"SUSPEND_CUSTOMER"}
Publishing {"id":"83bb14e7-9139-4699-a843-8b3a90ae26e2", □
    "type":"REINSTATE_CUSTOMER"}
Publishing {"id":"fe02fe96-a410-44a5-b636-dced53cf4590", □
    "firstName":"Bob","lastName":"Brown","type":"CREATE_CUSTOMER"}
Publishing {"id":"fe02fe96-a410-44a5-b636-dced53cf4590", □
    "firstName":"Charlie","lastName":"Brown","type":"UPDATE_CUSTOMER"}
```

When enough events have been emitted, stop the producer. Switch over to Kafdrop; you should see a series of JSON records appear in the `customer.test` topic:



Kafdrop — random customer events

Key and value deserializer

Analogously to the generic type constraints prevalent in the producer API, the `Consumer` interface enforces an equivalent constraint *vis-à-vis* the `ConsumerRecords` class returned by the `poll()` method, which carries a collection of individual `ConsumerRecord` objects:

```
/**
 * @see KafkaConsumer
 * @see MockConsumer
 */
public interface Consumer<K, V> extends Closeable {
  /** some methods omitted for brevity */

  /**
   * @see KafkaConsumer#poll(Duration)
   */
  ConsumerRecords<K, V> poll(Duration timeout);
}
```

```
/**
 * A container that holds the list {@link ConsumerRecord} per
 * partition for a particular topic. There is one
 * {@link ConsumerRecord} list for every topic partition returned
 * by a {@link Consumer#poll(java.time.Duration)} operation.
 */
public class ConsumerRecords<K, V>
    implements Iterable<ConsumerRecord<K, V>> {
  /** fields and methods omitted for brevity */
}
```

Similarly to the producer scenario, a consumer must be configured with the appropriate key and value deserializers. A deserializer must conform to the `org.apache.kafka.common.serialization.Deserializer` interface, listed below.

```
public interface Deserializer<T> extends Closeable {
  default void configure(Map<String, ?> configs, boolean isKey) {
    // intentionally left blank
  }

  T deserialize(String topic, byte[] data);

  default T deserialize(String topic, Headers headers, byte[] data) {
    return deserialize(topic, data);
  }

  @Override
  default void close() {
    // intentionally left blank
  }
}
```

The consumer client allows the user to specify the key and value deserializers in one of two ways:

1. Passing the fully-qualified class name of a `Deserializer` implementation to the consumer via the `key.deserializer` or the `value.deserializer` property.
2. Instantiating the deserializer and passing its reference to an overloaded `KafkaConsumer` constructor.

In virtually every way, the configuration of deserializers on the consumer is consistent with its producer counterpart. Behaviourally, deserializers are the logical reciprocal of serializers.

Akin to the serialization scenario, the user can select one of two strategies for unmarshalling data:

1. Implement a custom deserializer to directly handle the encoded form, such that the application code deals exclusively with typed payloads.
2. Piggyback on an existing deserializer, such as a `StringDeserializer` (for text encodings) or a `ByteArrayDeserializer` (for binary encodings), deferring the final unmarshalling of the encoded payload to the application.

There are no strong merits of one approach over the other that are worthy of a debate. Like in the producer scenario, we will use a custom deserializer to implement the forthcoming examples, being the idiomatic approach.

The section on serializers questioned the merits of type safety at the level of the producer, instead advocating for a façade over the top of the Kafka client code to insulate the business logic from the intricacies of Kafka connectivity, and to easily mock the latter in unit tests. As we will shortly discover, the case for a dedicated insulation layer is further bolstered when dealing with the consumer scenario.

Receiving events

Continuing from the producer example, let's examine the routine concerns of a typical business logic layer that might reside in a consumer application. How would it react to events received from Kafka? And more importantly, how would it even receive these events?

The standard mechanism for interacting with a Kafka consumer is to block on `Consumer.poll()`, then iterate over the returned records — invoking an application-level handler for each record. Kafka's defaults around automatic offset committing have also been designed specifically around this pattern — the *poll-process loop*.

The *poll-process loop* is a coined term, in lieu of an official name put forth by the Kafka documentation or a common name adopted by the user community.

A poll-process loop requires a thread on the consumer, as well as all the life-cycle management code that goes with it — when to start the thread, how to stop it, and so on.

Ideally, we would simply inject a high-level event receiver into the business logic, then register a listener callback with the receiver to be invoked every time the receiver pulls a record from the topic. Perhaps something along these lines:

```
public final class ConsumerBusinessLogic {
  private final EventReceiver receiver;

  public ConsumerBusinessLogic(EventReceiver receiver) {
    this.receiver = receiver;
    receiver.addListener(this::onEvent);
  }

  private void onEvent(CustomerPayload payload) {
    System.out.format("Received %s%n", payload);
  }
}
```

Again, what we do inside the business logic is of little consequence. The purpose of these examples is to illustrate how the business logic layer interacts with Kafka.

The `EventReceiver` and `EventListener` code listings, respectively:

```
public interface EventReceiver extends Closeable {
  void addListener(EventListener listener);

  void start();

  @Override
  void close();
}


@FunctionalInterface
public interface EventListener {
  void onEvent(CustomerPayload payload);
}
```

This approach completely decouples the `ConsumerBusinessLogic` class from the consumer code, being aware only of `EventReceiver`, which in itself is merely an interface. All communications with Kafka will be proxied via a suitable `EventReceiver` implementation.

Corrupt records

A producer has the benefit of knowing that the records given to it by the application code are valid, at least as far as the application is concerned. A consumer reading from an event stream does not have this luxury. A rogue or defective producer may have published garbage onto the topic, which would be summarily fed to all downstream consumers.

Ideally, we should handle any potential deserialization issues gracefully. As deserialization is within our control, we have several choices around the error-handling behaviour:

- Just log the error and discard the record;
- Propagate the error to the application via the (modified) callback, along with the malformed record; or
- Publish the malformed record to a dedicated 'dead letter' topic for subsequent inspection.

Assuming the decision is to pass the error to the application, the modified code might resemble the following:

```
public final class ConsumerBusinessLogic {
  public ConsumerBusinessLogic(EventReceiver receiver) {
    receiver.addListener(this::onEvent);
  }

  private void onEvent(ReceiveEvent event) {
    if (! event.isError()) {
      System.out.format("Received %s%n", event.getPayload());
    } else {
      System.err.format("Error in record %s: %s%n",
                        event.getRecord(), event.getError());
    }
  }
}


@FunctionalInterface
public interface EventListener {
  void onEvent(ReceiveEvent event);
}
```

The new `ReceiveEvent` class encapsulates both the `CustomerPayload` object — if one was unmarshalled successfully, or a `Throwable` error — if an exception occurred during unmarshalling. In both cases, the original `ConsumerRecord` is also included for reference, as well as the original encoded value. The source listing of `ReceiveEvent` follows.

```
public final class ReceiveEvent {
  private final CustomerPayload payload;

  private final Throwable error;

  private final ConsumerRecord<String, ?> record;

  private final String encodedValue;

  public ReceiveEvent(CustomerPayload payload,
                      Throwable error,
                      ConsumerRecord<String, ?> record,
                      String encodedValue) {
    this.record = record;
    this.payload = payload;
    this.error = error;
    this.encodedValue = encodedValue;
  }

  public CustomerPayload getPayload() {
    return payload;
  }

  public boolean isError() {
    return error != null;
  }

  public Throwable getError() {
    return error;
  }

  public ConsumerRecord<String, ?> getRecord() {
    return record;
  }

  public String getEncodedValue() {
    return encodedValue;
  }

  @Override
  public String toString() {
    return ReceiveEvent.class.getSimpleName() + " [payload="
        + payload + ", error=" + error + ", record=" + record
```

```
        + ", encodedValue=" + encodedValue + "]";
  }
}
```

The complete receiver

Now, to complete the implementation, we require a functioning `EventReceiver`. The listing below is that of the `DirectReceiver`, which is an implementation of the poll-process loop.

The choice of the term 'direct' is for consistency with the producer example. In both cases, the implementations directly employ the underlying Kafka API, without deviating from the standard behaviour or acquiring any additional characteristics — hence the name.

```
import java.time.*;
import java.util.*;

import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.errors.*;
import org.apache.kafka.common.serialization.*;

import com.obsidiandynamics.worker.*;

public final class DirectReceiver extends AbstractReceiver {
  private final WorkerThread pollingThread;

  private final Consumer<String, CustomerPayloadOrError> consumer;

  private final Duration pollTimeout;

  public DirectReceiver(Map<String, Object> consumerConfig,
                        String topic,
                        Duration pollTimeout) {
    this.pollTimeout = pollTimeout;

    final var mergedConfig = new HashMap<String, Object>();
    mergedConfig.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                     StringDeserializer.class.getName());
    mergedConfig.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                     CustomerPayloadDeserializer.class.getName());
    mergedConfig.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
                     false);
```

```
    mergedConfig.putAll(consumerConfig);
    consumer = new KafkaConsumer<>(mergedConfig);
    consumer.subscribe(Set.of(topic));

    pollingThread = WorkerThread.builder()
        .withOptions(new WorkerOptions()
                     .daemon()
                     .withName(DirectReceiver.class, "poller"))
        .onCycle(this::onPollCycle)
        .build();
  }

  @Override
  public void start() {
    pollingThread.start();
  }

  private void onPollCycle(WorkerThread t)
      throws InterruptedException {
    final ConsumerRecords<String, CustomerPayloadOrError> records;

    try {
      records = consumer.poll(pollTimeout);
    } catch (InterruptException e) {
      throw new InterruptedException("Interrupted during poll");
    }

    if (! records.isEmpty()) {
      for (var record : records) {
        final var payloadOrError = record.value();
        final var event =
            new ReceiveEvent(payloadOrError.getPayload(),
                             payloadOrError.getError(),
                             record,
                             payloadOrError.getEncodedValue());
        fire(event);
      }
      consumer.commitAsync();
    }
  }

  @Override
  public void close() {
```

```
    pollingThread.terminate().joinSilently();
    consumer.close();
  }
}
```

The `DirectReceiver` maintains a single polling thread. Rather than incorporating threading from first principles, the examples in this book use the *Fulcrum* micro-library, available at github.com/obsidiandynamics/f Specifically, the examples import the `fulcrum-worker` module, which provides complete life-cycle management on top of a conventional `java.lang.Thread`. A Fulcrum `WorkerThread` class provides an opinionated set of controls and templates that standardise all key aspects of a thread's behaviour — the initial startup, steady-state operation, interrupt-triggered termination, and exception handling. These are typical concerns in multi-threaded applications, which are difficult to get right and require a copious amount of non-trivial boilerplate code to adequately cover all the edge cases.

Our receiver takes three parameters — a map of configuration properties for the Kafka consumer, the name of the topic to subscribe to, and a timeout value to use in `Consumer.poll()`. Like its `DirectSender` counterpart, the `DirectReceiver` will overwrite certain key properties in the user-specified configuration map — settings that are required for the correct operation of the receiver and should not be interfered with by external code.

The `onPollCycle()` method represents a single iteration of the poll-process loop. Its role is straightforward — fetch records from Kafka, construct a corresponding `ReceiveEvent`, and dispatch the event to all registered listeners. Once all records in the batch have been dispatched, the `commitAsync()` method of the consumer is invoked, which will have the effect of asynchronously committing the offsets for all records fetched in the last call to `poll()`. Being asynchronous, the client will dispatch the request in a background thread, not waiting for the commit response from the brokers; the responses will arrive at an indeterminate time in the future, after `commitAsync()` returns.

The use of `commitAsync()` makes it possible to process multiple batches before the effects of committing the first are reflected on the brokers, increasing the window of uncommitted records. And while this will lead to a greater number of replayed records following partition reassignment, this behaviour is still consistent with the concept of *at-least-once* delivery. Using the blocking `commitSync()` variant reduces the number of uncommitted records to the in-flight batch at the expense of throughput. Unless the cost of processing a record is very high, the asynchronous commit model is generally preferred.

Finally, the `close()` method disposes of the receiver by terminating the polling thread, awaiting its termination, then closing the Kafka consumer.

Notice how we have caught an odd-looking `org.apache.kafka.common.errors.InterruptException` in the body of the `onPollCycle()` method, re-throwing a `java.lang.InterruptedException` in its place. This is one of the idiosyncrasies of the Kafka API — its origins are traceable to Scala,

[9] https://github.com/obsidiandynamics/fulcrum

which does not support checked exceptions, arguing vigorously against their use. As a result, the unchecked-exceptions-only philosophy has been carried over to the Java port, going against the grain of idiomatic Java.

```
try {
  records = consumer.poll(pollTimeout);
} catch (InterruptException e) {
  throw new InterruptedException("Interrupted during poll");
}
```

The standard Java thread interrupt signalling has been unceremoniously discarded in the bowels of the `KafkaConsumer` client and replaced with a bespoke runtime exception type. The code above corrects for this, trapping the bespoke exception and re-throwing a standard one. When this occurs, the Fulcrum `WorkerThread` will detect the interrupt and gracefully shut down the underlying primordial thread.

To run the example, launch the `RunDirectConsumer` class:

```
public final class RunDirectConsumer {
  public static void main(String[] args)
      throws InterruptedException {
    final Map<String, Object> consumerConfig =
        Map.of(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9092",
               ConsumerConfig.GROUP_ID_CONFIG,
               "customer-direct-consumer",
               ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
               "earliest");

    try (var receiver = new DirectReceiver(consumerConfig,
                                           "customer.test",
                                           Duration.ofMillis(100))) {
      new ConsumerBusinessLogic(receiver);
      receiver.start();
      Thread.sleep(10_000);
    }
  }
}
```

This will run for ten seconds, outputting the records that have been published since the consumer was last run. Since we are running it for the first time, expect to see all records in the `customer.test` topic:

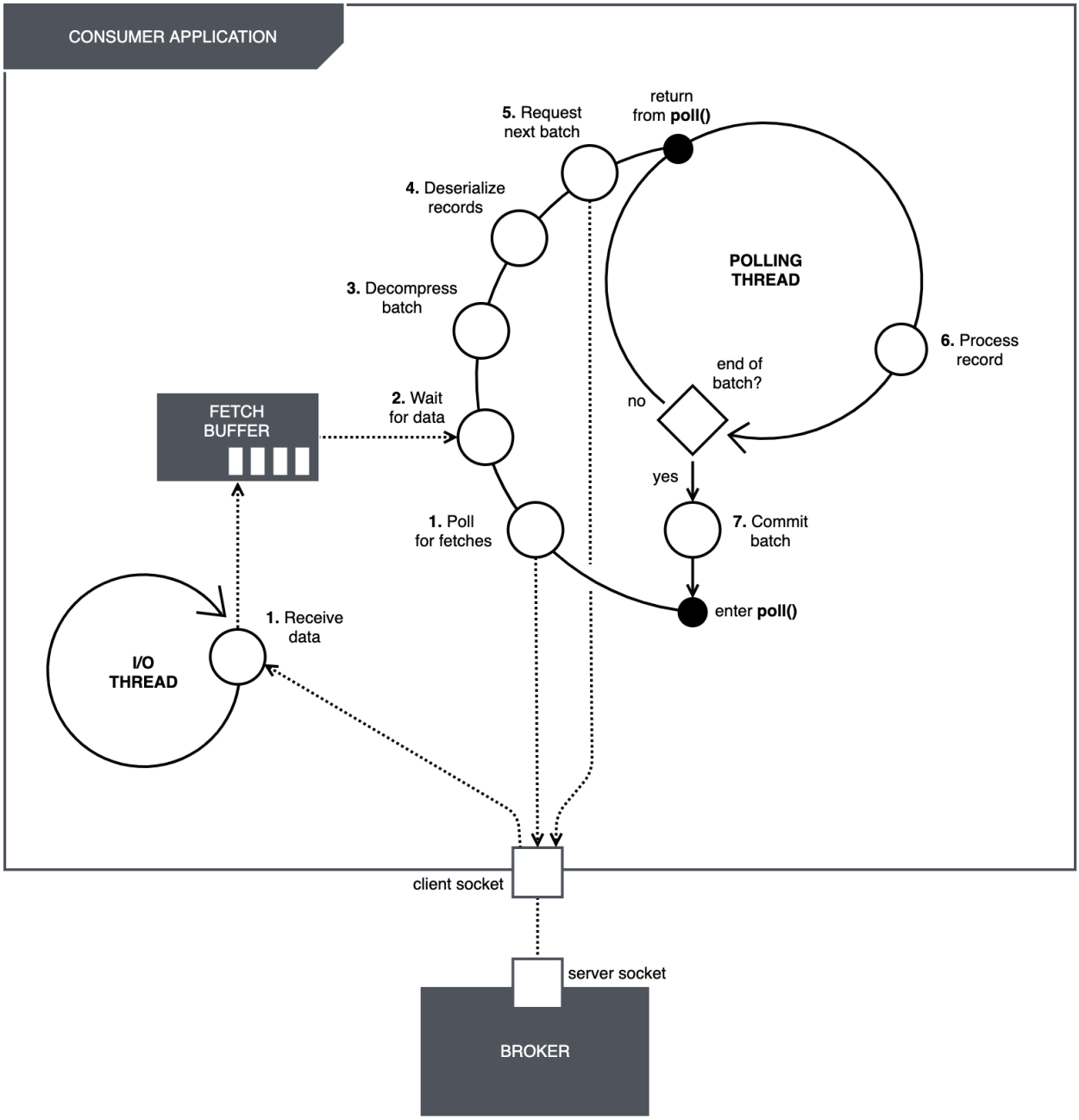
```
...
omitted for brevity
...
Received CreateCustomer [id=28361a15-7cef-47f3-9819-f8a629491c5a, ▯
    firstName=Bob, lastName=Brown]
Received SuspendCustomer [id=28361a15-7cef-47f3-9819-f8a629491c5a]
Received ReinstateCustomer [id=28361a15-7cef-47f3-9819-f8a629491c5a]
Received CreateCustomer [id=b3cd538c-90cc-4f5f-a5dc-b1c469fc0bf8, ▯
    firstName=Bob, lastName=Brown]
Received UpdateCustomer [id=b3cd538c-90cc-4f5f-a5dc-b1c469fc0bf8, ▯
    firstName=Charlie, lastName=Brown]
Received CreateCustomer [id=83bb14e7-9139-4699-a843-8b3a90ae26e2, ▯
    firstName=Bob, lastName=Brown]
Received UpdateCustomer [id=83bb14e7-9139-4699-a843-8b3a90ae26e2, ▯
    firstName=Charlie, lastName=Brown]
Received SuspendCustomer [id=83bb14e7-9139-4699-a843-8b3a90ae26e2]
Received ReinstateCustomer [id=83bb14e7-9139-4699-a843-8b3a90ae26e2]
Received CreateCustomer [id=fe02fe96-a410-44a5-b636-dced53cf4590, ▯
    firstName=Bob, lastName=Brown]
Received UpdateCustomer [id=fe02fe96-a410-44a5-b636-dced53cf4590, ▯
    firstName=Charlie, lastName=Brown]
```

Pipelining

One material argument for layering a consumer application stems from the realm of performance optimisation, namely a technique called ‘pipelining’.

A *pipeline* is a decomposition of a sequential process into a set of chained stages, where the output of one stage is fed as the input to the next via an intermediate bounded buffer. Each stage functions semi-independently of its neighbours; it can operate for as long as at least one element is available in its input buffer and will also halt for as long as the output buffer is full.

Pipelining allows the application to recruit additional threads — increasing the throughput at the expense of processor utilisation. To appreciate where the performance gains may be obtained, consider the routine operation of a regular consumer application — identical to the one we just implemented — illustrated below.



Consumer without additional pipelining

The `KafkaConsumer` implementation utilises a rudimentary form of pipelining under the hood, prefetching and buffering records to accelerate content delivery. In other words, our application is already multi-threaded with us hardly realising this — separating the record retrieval and processing operations into distinct execution contexts. The main *poller* thread will —

1. Invoke `KafkaConsumer.poll()`, potentially sending fetch queries to the cluster. If there are pending queries for which responses have not yet been received, no further queries are issued.

2. Wait for the outcome of a pending fetch, checking the status of the fetch buffer. The accumulation of the fetch results will be performed by a background I/O thread. This operation will block until the data becomes available or the poll timeout expires.
3. Decompress the batch if compression was set on the producer.
4. Deserialize each record in the batch.
5. Prior to returning from `poll()`, initiate a prefetch. This action is non-blocking; it fires off a set of queries to the brokers and returns immediately, without waiting for responses. The control is transferred back to the application code. When the prefetch responses eventually arrive, these will be decompressed and deserialized, with the resulting records placed into a fetch buffer.
6. The application then applies the requisite business logic to each record by invoking the registered `EventListener` callbacks. In most applications, this would involve updating a database and possibly other I/O. More often than not, the cost of processing a record is significantly greater than the cost of reading it off a Kafka topic.
7. After the application has completed processing the batch, it can safely commit the consumer's offsets by invoking `Consumer.commitAsync()`. This will have the effect of committing the offsets for all records returned during the last `poll()`. Being an asynchronous operation, the committing of offsets will occur in a background I/O thread.

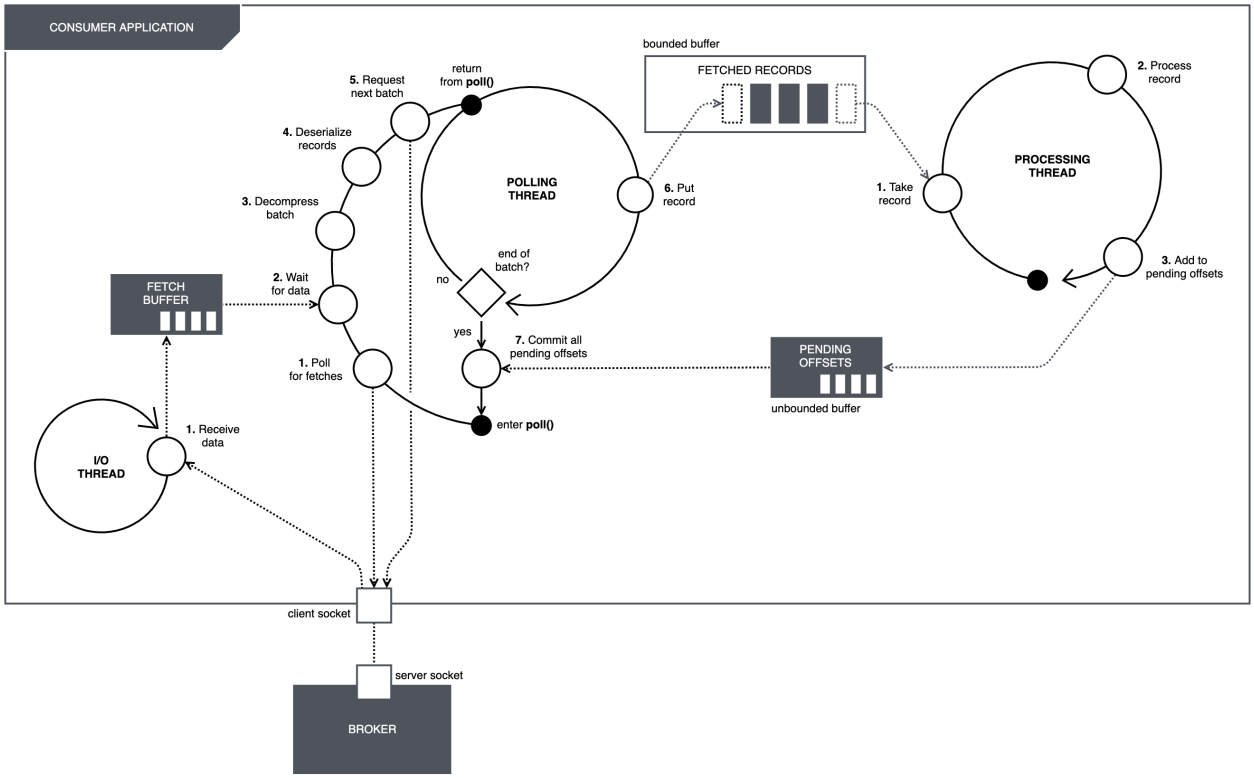
The last step in the poll-process loop is optional, in that we could have just deferred to the consumer's built-in 'automatic offset committing' feature by leaving `enable.auto.commit` at its default value of `true`. In the `DirectReceiver` example, we have chosen to disable offset auto-commit, going with the manual option instead. The principal advantage of committing offsets manually is that it results in a narrower window of uncommitted offsets compared to the automatic option, even if done asynchronously, as the auto-commit is bounded by a timer and lazily initiated. Conversely, the benefit of the offset auto-commit feature is the reduction in the number of commit requests sent to the brokers, which has a small positive effect on throughput.

The I/O thread works in the background. Among its chief responsibilities is the handling of responses from earlier fetch requests, accumulating the received batches in a fetch buffer. With the assistance of the background prefetch mechanism, and assuming a steady flow of records through the pipeline, most calls to `poll()` should not block.

While the `KafkaConsumer` allows for pipelining via its prefetch mechanism, the implementation stacks the deserialization of records and their subsequent handling onto a single thread of execution. The thread that is responsible for deserializing the records is also used to drive business logic. Both operations are potentially time-consuming; when a record is being deserialized, the polling thread is unable to execute the `EventListener` callbacks, and vice versa.

While we cannot control this element of the client's standard behaviour, we can make greater use of the pipeline pattern, harnessing additional performance gains by separating record deserialization from payload handling.

The diagram below illustrates this.



Consumer with pipelining

The fetching, deserialization, and processing of records has now been separated into three stages, each powered by a dedicated thread. For simplicity, we are going to refer to these as the *I/O thread*, the *polling thread*, and the *processing thread*. The I/O thread is native to the `KafkaConsumer` and its behaviour is unchanged from the previous example.

The polling thread is altered in two crucial ways:

1. Rather than invoking the `EventListener` in step 6, the thread will append the received record onto a bounded buffer. In a Java application, we can use an `ArrayBlockingQueue` or a `LinkedBlockingQueue` to implement this buffer. This operation will block if the queue is at its maximum capacity.
2. Instead of committing the offsets of the recent batch, the polling thread will commit just those offsets that have been appended to the 'pending offsets queue' by the processing thread.

On the processing thread, we have the following steps:

1. Remove the queued record from the bounded buffer. This operation will block if the queue is empty.
2. Invoke the registered `EventListener` callbacks to process the record.
3. Having processed the record, append a corresponding entry to the 'pending offsets queue'. This entry specifies the topic-partition pair for the record, as well as its offset *plus one*.

The last steps in each of the two threads may appear confusing at first. Couldn't we just commit the offsets after enqueuing the batch? What is the purpose of shuttling the offsets back from the processing thread to the polling thread? And why would we add one to the offset of a processed record?

When pipelining records, one needs to take particular care when committing the records' offsets, as the records might not be processed for some time after being queued. Depending on the capacity of the bounded buffer, the polling loop may complete several cycles before the processing thread gets an opportunity to attend to the first queued record. The failure of the consumer application would lead to missed records; the newly assigned consumer will have naturally assumed that the committed records were processed. To achieve at-least-once delivery semantics, the offsets of a record must be committed at some point after the record is processed.

While disabling `enable.auto.commit` is optional in the direct consumer scenario, it must categorically be disabled in the pipeline scenario. The effect of leaving offset auto-commit on is the logical equivalent of committing records after queuing them, with no regard as to whether they were processed by the downstream stage.

The need to shuttle the offsets back to the I/O thread addresses an inherent limitation of the `KafkaConsumer` implementation. Namely, *the consumer is not thread-safe*. Attempting to invoke `commitAsync()` from a thread that is different to the one that invoked `poll()` will result in a `java.util.ConcurrentModificationException` exception. As such, we have no choice but to repatriate the offsets to the polling thread.

The final point — the addition of one to a record's offset — accounts for the fact that a Kafka consumer will start processing records from the exact offset persisted against its encompassing consumer group. Naively committing the record's offset 'as is' will result in the replaying of the last committed record following a topic rebalancing event, where partitions may be reassigned among the consumer population. By adding one to the offset, we are ensuring that the new assignee will skip over the last processed record — seamlessly taking over from where the last consumer left off.

Invoking the no-argument `commitAsync()` method, as in the `DirectReceiver` scenario, will automatically add one to the offsets of the last records processed for each partition. When specifying offsets explicitly, the offset arithmetic becomes the responsibility of the application.

For our next trick, we shall conjure up an alternate receiver implementation, this time exploiting the pipeline pattern outside of the consumer:

```
import java.time.*;
import java.util.*;
import java.util.concurrent.*;

import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.*;
import org.apache.kafka.common.errors.*;
import org.apache.kafka.common.serialization.*;

import com.obsidiandynamics.worker.*;
import com.obsidiandynamics.worker.Terminator;

public final class PipelinedReceiver extends AbstractReceiver {
  private final WorkerThread pollingThread;

  private final WorkerThread processingThread;

  private final Consumer<String, CustomerPayloadOrError> consumer;

  private final Duration pollTimeout;

  private final BlockingQueue<ReceiveEvent> receivedEvents;

  private final Queue<Map<TopicPartition, OffsetAndMetadata>>
      pendingOffsets = new LinkedBlockingQueue<>();

  public PipelinedReceiver(Map<String, Object> consumerConfig,
                           String topic,
                           Duration pollTimeout,
                           int queueCapacity) {
    this.pollTimeout = pollTimeout;
    receivedEvents = new LinkedBlockingQueue<>(queueCapacity);

    final var mergedConfig = new HashMap<String, Object>();
    mergedConfig.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                     StringDeserializer.class.getName());
    mergedConfig.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                     CustomerPayloadDeserializer.class.getName());
    mergedConfig.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
                     false);
    mergedConfig.putAll(consumerConfig);
    consumer = new KafkaConsumer<>(mergedConfig);
    consumer.subscribe(Set.of(topic));
```

```
    pollingThread = WorkerThread.builder()
        .withOptions(new WorkerOptions()
                     .daemon()
                     .withName(PipelinedReceiver.class, "poller"))
        .onCycle(this::onPollCycle)
        .build();

    processingThread = WorkerThread.builder()
        .withOptions(new WorkerOptions()
                     .daemon()
                     .withName(PipelinedReceiver.class, "processor"))
        .onCycle(this::onProcessCycle)
        .build();
  }

  @Override
  public void start() {
    pollingThread.start();
    processingThread.start();
  }

  private void onPollCycle(WorkerThread t)
      throws InterruptedException {
    final ConsumerRecords<String, CustomerPayloadOrError> records;

    try {
      records = consumer.poll(pollTimeout);
    } catch (InterruptException e) {
      throw new InterruptedException("Interrupted during poll");
    }

    if (! records.isEmpty()) {
      for (var record : records) {
        final var value = record.value();
        final var event = new ReceiveEvent(value.getPayload(),
                                           value.getError(),
                                           record,
                                           value.getEncodedValue());
        receivedEvents.put(event);
      }
    }
```

```
    for (Map<TopicPartition, OffsetAndMetadata> pendingOffset;
        (pendingOffset = pendingOffsets.poll()) != null;) {
      consumer.commitAsync(pendingOffset, null);
    }
  }

  private void onProcessCycle(WorkerThread t)
      throws InterruptedException {
    final var event = receivedEvents.take();
    fire(event);
    final var record = event.getRecord();
    pendingOffsets
    .add(Map.of(new TopicPartition(record.topic(),
                                   record.partition()),
                new OffsetAndMetadata(record.offset() + 1)));
  }

  @Override
  public void close() {
    Terminator.of(pollingThread, processingThread)
    .terminate()
    .joinSilently();
    consumer.close();
  }
}
```

There are a few notable differences between a `PipelinedReceiver` and its `DirectReceiver` counterpart:

1. The addition of a second thread — we now have a distinct `processingThread` and a `pollingThread`, whereas the original implementation made do with a single `pollingThread`. Correspondingly, the pipelined implementation has two `onCycle` handlers.
2. The addition of a `LinkedBlockingQueue`, acting as a bounded buffer between the two worker threads.
3. An additional parameter to the constructor, specifying the capacity of the blocking queue.

The `onPollCycle()` method fetches records, but does not dispatch the event. Instead, it puts the event onto the `receivedEvents` queue, blocking if necessary until space becomes available. Having dealt with the batch, it will gather any pending offsets that require committing, taking care not to block while consuming from the `pendingOffsets` queue.

Using the non-blocking `Queue.poll()` method prevents a deadlock condition, where the polling thread is blocked on the processor thread to submit additional offsets, while the processor thread is hopelessly waiting on the polling thread to convey records through the pipeline.

The `onProcessCycle()` method takes records from the head of the `receivedEvents` queue, waiting if necessary for an event to become available. The event is then dispatched to all registered listeners. Finally, the offsets of the underlying record are incremented and submitted to the `pendingOffsets` queue for subsequent committing by the polling thread.

To run the pipelined example, launch the `RunPipelinedConsumer` class:

```
public final class RunPipelinedConsumer {
  public static void main(String[] args)
      throws InterruptedException {
    final Map<String, Object> consumerConfig =
        Map.of(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9092",
               ConsumerConfig.GROUP_ID_CONFIG,
               "customer-pipelined-consumer",
               ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
               "earliest");

    try (var receiver =
             new PipelinedReceiver(consumerConfig,
                                   "customer.test",
                                   Duration.ofMillis(100), 10)) {
      new ConsumerBusinessLogic(receiver);
      receiver.start();
      Thread.sleep(10_000);
    }
  }
}
```

By applying the pipeline pattern, we have decoupled two potentially slow operations, allowing them to operate independently of one another. The performance gains are not exclusive to multi-core or multi-processor architectures; even single-core, pseudo-concurrent systems will benefit from pipelining by maximising the amount of useful work a processor can do.

One of the perceived drawbacks of pipelining is that it adds latency to the process; the contention over a shared buffer and the overhead of thread scheduling and cache coherence will add to the end-to-end propagation delay of a record flowing through the pipeline. And while the added latency is typically more than made up for in throughput gains, it is really up to the application designer to

make the final call on the optimisation strategy. An argument could be made that since *Kafka-based applications tend to be throughput-oriented*, optimisations of this nature are timely and appropriate.

Kafka's performance doctrine has traditionally been to add more consumers, more partitions, and more brokers. Often, little regard is given to the cost-effectiveness of this model. It is not uncommon to observe a high number of mostly-idling consumers spawned in an autoscaling group to get through a moderately loaded topic. One cannot help but question the actual number of consumer instances required, had each instance utilised its available resources to their provisioned capacity.

And now for the best part. Previous examples have leveraged the `EventReceiver` interface to decouple the business logic from the low-level Kafka `Consumer` handling. As we have just demonstrated, pipelining can fit entirely into the `EventReceiver` implementation, with *no impact to the business logic layer*. The complexities of multi-threaded code, queuing of records, and the shuttling of offsets are entirely concealed behind the `EventReceiver` interface. In fact, we can stock multiple `EventReceiver` implementations — a more conventional `DirectReceiver` and a multi-threaded `PipelinedReceiver`. These are functionally equivalent, but exhibit different performance and resource utilisation characteristics. The `PipelinedReceiver` adds 30 or so lines on top of the `DirectReceiver` implementation. Comparatively speaking, this may seem like a lot, given that the extra code is around 50% of the original implementation. But in the larger scheme of things, 30 lines are small potatoes and the added complexity is incurred *once* — the improved resource utilisation profile and the resulting performance gains more than make up for the additional effort.

In case the reader is wondering, the pipelining optimisation is less effective in producer applications, as serialization is typically much less processor and memory-intensive compared to deserialization. Also, the `KafkaProducer` is pipelined internally, separating serialization from network I/O. Little in the way of performance gains would be accomplished by moving record serialization to a dedicated thread.

Record filtering

In rounding off this chapter, we shall highlight another compelling reason for an abstraction layer: the filtering of records. Filtering fulfills a set of use cases where either a deserializer, or an application-level unmarshaller might conditionally present a record to the rest of the application. This is not a native capability of a Kafka consumer, requiring a bespoke implementation.

The natural question one might ask is: Why not filter at the business logic layer with some `if` statements?

There are two challenges with this approach, which also become more difficult to solve as one moves up the application stack. Firstly, it assumes that the client has the requisite domain objects that can be mapped from a record's serialized form. Secondly, it incurs the performance overhead of unconditionally unmarshalling all records, only to discard some records shortly thereafter.

The first problem — missing domain objects or the lack of knowledge of certain record schemas (the two are logically equivalent) — can be attributed to several causes:

- The source Kafka topic is broadly-versed, containing more types of records that the consumer legitimately requires for routine operation.
- The record types might be known to the consumer, but it may have no interest in processing them. For example, a topic might represent changes to a global customer database, whereas a consumer deployed in a single region might only care about the subset of customers in its locality.
- The record structure has evolved over time, such that the topic may contain records that comply to varying schema versions. To accommodate the gradual transition of consumers to a newer schema, producers will typically publish the same record in multiple versions.

At any rate, the consumer will have to selectively parse the record's value on the basis of some explicit indicator. This indicator may be a well-known header — for example, `recordType: CREATE_CUSTOMER`, `region: Asia-Pacific`, or `version: 2`. Alternatively, the indicator may be inferred from the record's value without having to parse the entire payload, let alone mapping the payload to a POJO. For example, the Jackson library allows you to create a custom deserializer that can inspect the document object model before deciding to map it to an existing Java class.

In some cases, filtering records from within the business logic is a valid approach, particularly where the filtering predicates require deep inspection of the record's payload or are related to the current application state.

The final point relates to performance. While premature optimisation should be avoided, there may be legitimate cases where the sheer amount of data on a topic places a strain on the consumer ecosystem, particularly when the topic is coarse-grained. If the options for increasing the granularity of topics and the scaling of consumers have been exhausted, the sole remaining option might be to pre-filter records in an attempt to extract the last ounce of performance. Sounds terrible? Agreed! The recommendation is to avoid complexity on the basis of performance alone. Instead, the application should be architected from the outset to cope with the expected load.

Kafka's idiomatic approach for dealing with varying record representations is through custom (de)serializers. Once implemented and configured, (de)serializers work behind the scenes, accepting and delivering application-native record keys and values via a generically typed API. The producer application is expected to address the `Producer` implementation directly, while on the consumer-end, this approach is often paired with a simple poll-process loop.

This chapter has explored some of the typical concerns of producer and consumer applications, arguing for the use of an abstraction layer to separate Kafka-specific messaging code from the

business logic. This makes it easier to encapsulate common behaviour and invariants on one hand, and on the other, simplifies key aspects of the application, making it easier to mock and test in isolation.

We have also come to understand the inefficiency inherent in the poll-process loop, namely the stacking of record deserialization and processing onto a single execution thread. The internal pipelining model of a `KafkaConsumer` was explained, and we explored how the concept of pipelining can be exploited to further decouple the deserialization of records from their subsequent processing.

Finally, we have looked at record versioning and filtering as prime use cases for concealing non-trivial behaviour behind an abstraction layer, reducing the amount of work that needs to happen at the processing layer and its resulting complexity.

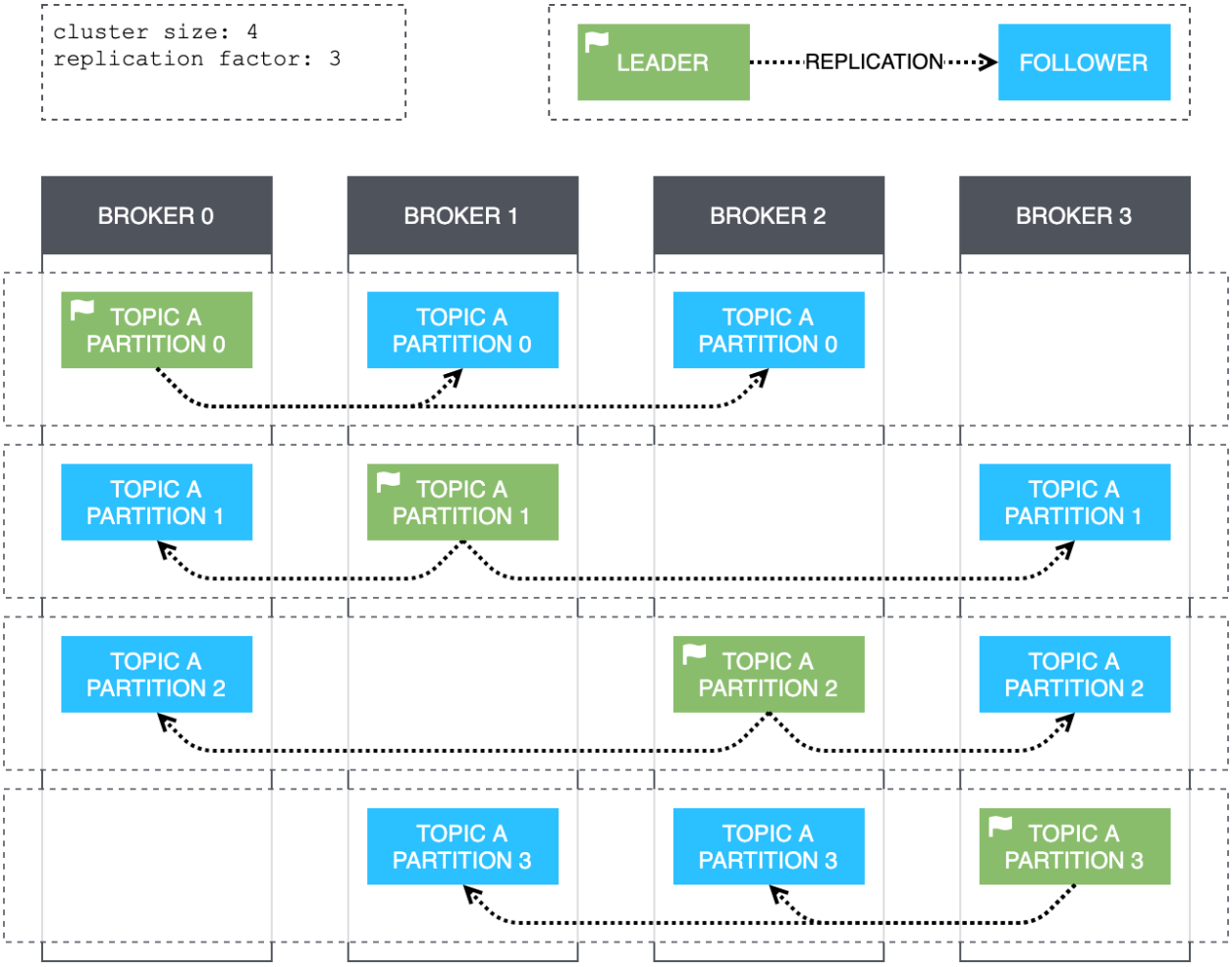
Chapter 8: Bootstrapping and Advertised Listeners

Who are these listeners? And what are they advertising?

Having been in the Kafka game since 2015, without exaggeration, the most common question that gets asked is: "Why can't I connect to my broker?" And it is typically followed up with: "I'm sure the firewall is open; I tried pinging the box; I even tried telnetting into port 9092, but I still can't connect." The bootstrapping configuration will frustrate the living daylights out of most developers and operations folk at some point, and that's unfortunate; for all the flexibility that Kafka has to offer, it certainly isn't without its drawbacks.

A gentle introduction to bootstrapping

Before we can start looking into advertised listeners, we need a thorough understanding of the client bootstrapping process. As it was previously stated, Kafka replicates a topic and its underlying partitions among several broker nodes, such that one broker will act as a leader for one partition and a follower for several others. Assuming that a topic has many partitions and the allocation of replicas is approximately level, no single broker will master a topic in its entirety. The illustration below depicts a four-broker cluster hosting a topic with four partitions, with a replication factor of three. (Meaning that each partition will have one leader and two follower replicas.)



Topic replication

Now let's take the client's perspective for a moment. When a producer wishes to publish a record, it must contact the lead broker for the target partition, which in turn, will disseminate the record to the follower replicas, before acknowledging the write. Since a producer will typically publish on all partitions at some point, it will require a *direct* connection to most, if not all, brokers in a Kafka cluster. (Whether these connections are established *eagerly* — at initialisation, or *lazily* — on demand, will largely depend on the client library implementation.)

Unfortunately Kafka brokers are incapable of forwarding a write request to the lead broker. When attempting to publish a record to a broker that is not a declared partition leader within a replica set, the latter will fail with a `NOT_LEADER_FOR_PARTITION` error. This error sits in the category of *retryable errors*, and may occur from time to time, notably when the leadership status transitions among the set of in-sync replicas for whatever reason. The producer client will simply fetch the new cluster metadata, discover the updated leader, and will re-address the write request accordingly. The `NOT_LEADER_FOR_PARTITION` error is mostly harmless in small doses; however, repeated and unresolved recurrence of this error suggests a more profound problem.

The challenge boils down to this: How does a client discover the nodes in a Kafka cluster? A naive solution would have required us to explicitly configure the client with the complete set of individual addresses of each broker node. Establishing direct connections would be trivial, but the solution would not scale to a dynamic cluster topology; adding or removing nodes from the cluster would require a reconfiguration of all clients.

The solution that Kafka designers went with is based on a *directory* metaphor. Rather than being told the broker addresses, clients look up the *cluster metadata* in a directory in the first phase in the bootstrapping process, then establish direct connections to the discovered brokers in the second phase. Rather than coming up with more moving parts, the role of the directory is conveniently played by each of the brokers. Since brokers are intrinsically aware of one another via ZooKeeper, every broker has an identical view of the cluster metadata, encompassing every other broker, and is able to impart this metadata onto a requesting client. Still, the brokers might change, which seemingly contradicts the notion of a 'stable' directory. To counteract this, clients are configured with a *bootstrap list* of broker addresses that only needs to be partially accurate. As long as one address in the bootstrap list points to a *live* broker, the client will learn the entire topology.

Taking advantage of DNS

You would be right in thinking that this model feels brittle. What if we recycle all brokers in a cluster? What if the brokers are hosted on ephemeral instances in the Cloud and may come and go as they please, with a new IP address each time? The bootstrap list would soon become useless. Aren't we only kicking the 'reconfiguration can' down the road?

While there is no *official* response to this, the practice adopted in the community is to use a second tier of DNS entries. Suppose we had an arbitrarily-sized cluster that could be recycled on demand. Each broker would be assigned an IP address and likely an auto-generated hostname, both being ephemeral. To complement the directory metaphor, we would create a handful of well-known *canonical* DNS `CNAME` or `A` records with a minimal TTL, pointing to either the IP addresses or the hostnames of a subset of our broker nodes. The fully-qualified domain names of new DNS entries might be something like —

- `broker0.ext.prod.kafka.mycompany.com`.
- `broker1.ext.prod.kafka.mycompany.com`.
- `broker2.ext.prod.kafka.mycompany.com`.

The Kafka clients would only be configured with the list of canonical bootstrap addresses. Every address-impacting broker change would entail updating the canonical DNS entries; but it's easier to keep DNS up to date, then to ensure that all clients (of which there could be hundreds or thousands) are correctly configured. Furthermore, DNS entries can easily be tested. One could implement a trivial 'canary' app that periodically attempts to connect to the brokers on one of the canonical addresses to verify their availability. This way we would soon learn when something untoward happens, before the issue escalates to the point of an all-out failure.

An elaboration of the above technique is to use round-robin DNS. Rather than maintaining multiple `A` records for unique hosts, DNS permits several `A` records for the same host, pointing to different IP addresses. A DNS query for a host will return all matching `A` records, permuting the records prior to returning. Furthermore, the returned list may be a mixture of IPv4 and IPv6 addresses. Assuming the client will try the first IP address in the returned list, each address will serve an approximately equal number of requests. The client does not have to stop at the first entry; it can try any number of them, potentially *all* of them, until it reaches a host that is able to satisfy its request.

The ability to utilise all resolved addresses was introduced to Kafka in version 2.1, as part of KIP-302[10]. (*KIP* stands for *Kafka Improvement Proposal*.) To maintain backward-compatible behaviour, Kafka disables this by default. To enable this feature, set the `client.dns.lookup` configuration to `use_all_dns_ips`. Once enabled, the client will utilise all resolved DNS entries.

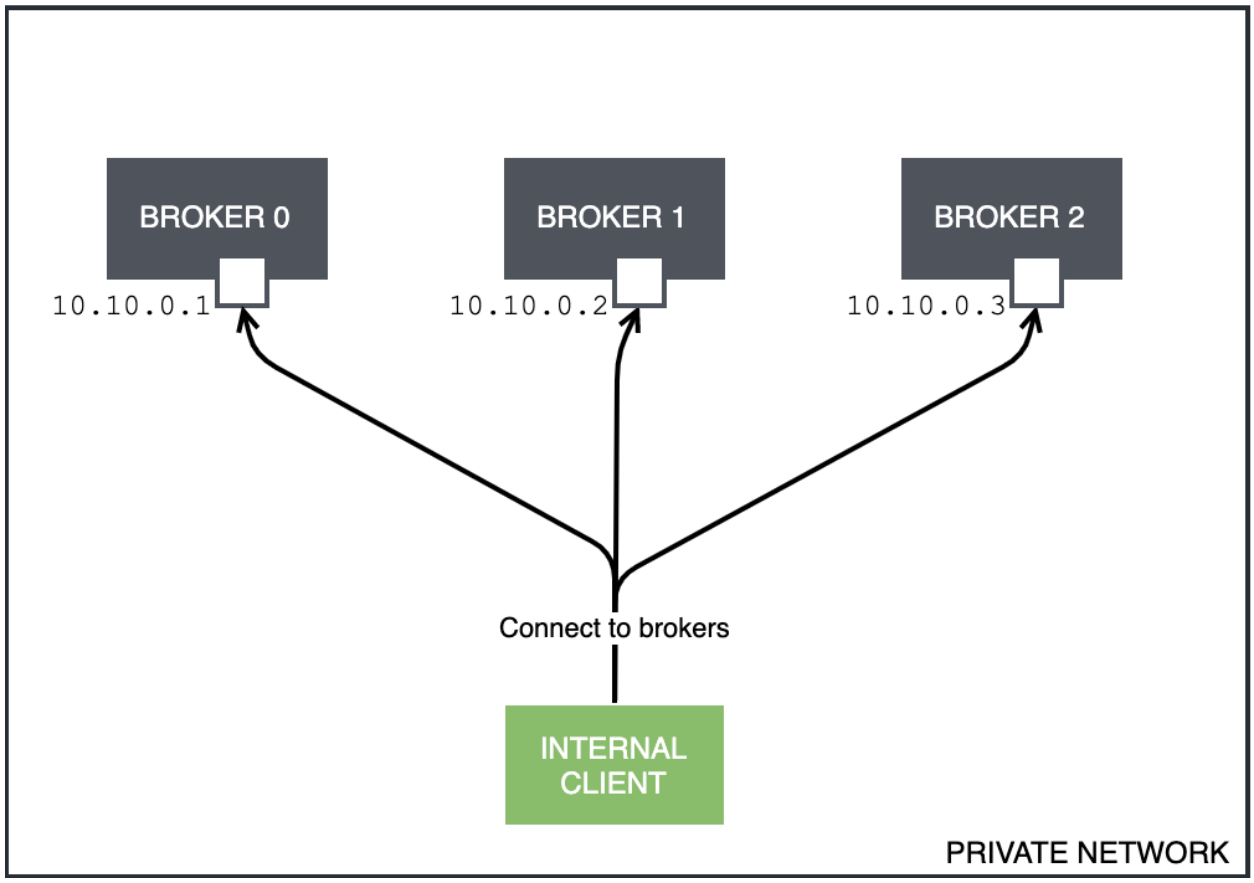
The advantage of this approach is that it does not require us to alter the bootstrap list when adding more fallback addresses. The list may be reduced to a single entry — for example, `broker.ext.prod.kafka.mycompany` — which will resolve to an arbitrary number of IP addresses, depending on how the DNS records are configured. Furthermore, multi-record DNS resolution applies not only to bootstrapping, but also to subsequent connections to the resolved brokers.

The use of multiple alternate records for the same host requires the resolved brokers to agree on a common port number. Also, it is limited to `A` records; the DNS specification forbids the use of multiple `CNAME` records for the same fully-qualified domain name. Another *potential* limitation relates to the client implementation. The `client.dns.lookup` property is accepted by the Java client library; ports to other languages might not support this capability — check with your library before using this feature. With the exception of the last point, these constraints are rarely show-stoppers in practice. The benefit of this approach — having a centralised administration point with a set-and-forget bootstrap list — may not be immediately discernible with a handful of clients, but becomes more apparent as the client ecosystem grows.

A simple scenario

In a simple networking topology, where each broker can be reached on a single address and port number, the bootstrapping mechanism can be made to work with minimal configuration. Consider a simple scenario with three brokers confined to a private network, such that the producer and consumer clients are also deployed on the same network. Keeping things simple, let's assume the broker IP addresses are `10.10.0.1`, `10.10.0.2`, and `10.10.0.3`. Each broker is listening on port `9092`. A client application deployed on `10.20.0.1` is attempting to connect to the cluster. This scenario is illustrated below.

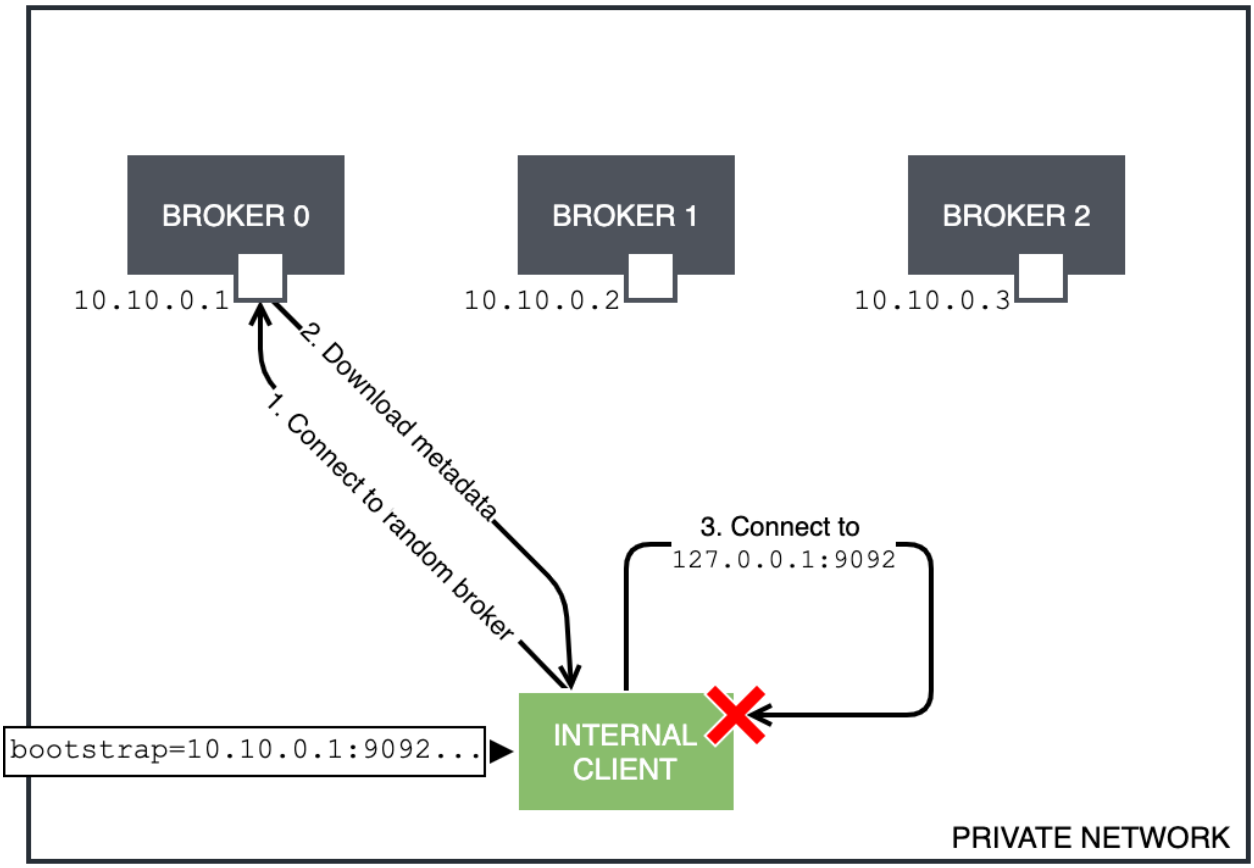
[10] https://cwiki.apache.org/confluence/display/KAFKA/KIP-302+-+Enable+Kafka+clients+to+use+all+DNS+resolved+IP+addresses



Connecting within a private network

At this point one would naturally assume that passing in `10.10.0.1:9092,10.10.0.2:9092,10.10.0.3:9092` for the bootstrap list should just work. The problem is that the Kafka broker does not know which IP address or hostname it should advertise, and it does a pretty bad job at auto-discovering this. In most cases, it will default to `localhost`.

The diagram below captures the essence of the problem. Upon bootstrapping, the client will connect to `10.10.0.1:9092`, being the first element in the bootstrap list. (In practice, the client will pick an address at random, but it hardly matters in this example.) Having made the connection, the client will receive the cluster metadata — a list of three elements — each being `localhost:9092`. You can see where this is going. The client will then try connecting to `localhost` — to itself. *Et voila*, that is how the dreaded *"Connection to node -1 (localhost/127.0.0.1:9092) could not be established. Broker may not be available."* error is obtained.



Internal client looping back to itself

This is as much of a problem for simple single-broker Kafka installations as it is for multi-broker clusters. Clients will always follow addresses revealed by the cluster metadata even if there is only one node.

This is solved with *advertised listeners*. (Finally, we are getting around to the crux of the matter.) A Kafka broker may be configured with three properties — `advertised.listeners`, `listeners`, and `listener.security.protocol.map` — which are interrelated and designed to be used in concert. Let's open the broker configuration and find these properties. Edit `$KAFKA_HOME/config/server.properties`; towards the beginning of the file you should see several lines resembling the following:

```
################## Socket Server Settings ####################

# The address the socket server listens on. It will get the value
# returned from java.net.InetAddress.getCanonicalHostName() if
# not configured.
#   FORMAT:
#     listeners = listener_name://host_name:port
#   EXAMPLE:
#     listeners = PLAINTEXT://your.host.name:9092
#listeners=PLAINTEXT://:9092

# Hostname and port the broker will advertise to producers and
# consumers. If not set, it uses the value for "listeners" if
# configured.  Otherwise, it will use the value returned from
# java.net.InetAddress.getCanonicalHostName().
#advertised.listeners=PLAINTEXT://your.host.name:9092

# Maps listener names to security protocols, the default is for
# them to be the same. See the config documentation for more details
#listener.security.protocol.map=PLAINTEXT:PLAINTEXT,\
    SSL:SSL,SASL_PLAINTEXT:SASL_PLAINTEXT,SASL_SSL:SASL_SSL
```

Unless you have previously changed the configuration file, the properties will start off commented out. The `listeners` property is structured as a comma-separated list of URIs, which specify the sockets that the broker should listen on for incoming TCP connections. Each URI comprises a free-form protocol name, followed by a `://`, an optional interface address, followed by a colon, and finally a port number. Omitting the interface address will bind the socket to the default network interface. Alternatively, you can specify the `0.0.0.0` meta-address to bind the socket on all interfaces.

An interface address may also be referred to as a *bind address*.

In the default example, the value `PLAINTEXT://:9092` can be taken to mean a listener protocol named `PLAINTEXT`, listening on port `9092` bound to the default network interface.

A commented-out property will assume its default value. Defaults are listed in the official Kafka documentation page at kafka.apache.org/documentation[11], under the 'Broker Configs' section.

The protocol name must map to a valid security protocol in the `listener.security.protocol.map` property. The security protocols are fixed, constrained to the following values:

[11] https://kafka.apache.org/documentation/#brokerconfigs

- `PLAINTEXT`: Plaintext TCP connection without user principal authentication.
- `SSL`: TLS connection without authentication.
- `SASL_PLAINTEXT`: Plaintext connection with SASL (Simple Authentication and Security Layer) to authenticate user principals.
- `SASL_SSL`: The combination of TLS for transport-level security and SASL for user principal authentication.

To map a listener protocol to a security protocol, uncomment the `listener.security.protocol.map` property and append the new mapping. The mapping must be in the form `<listener protocol name>:<security protocol name>`.

The protocol name is a major source of confusion among first-time Kafka users. There is a misconception, and unsurprisingly so, that `PLAINTEXT` at the `listener` level relates to the encryption scheme (or lack thereof). In fact, the listener protocol names are completely arbitrary. And while it is good practice to assign them meaningful names, a listener URI `DONALD_DUCK://:9092` is perfectly valid, provided that `DONALD_DUCK` has a corresponding mapping in `listener.security.protocol.map`, e.g. `DONALD_DUCK:SASL_SSL`.

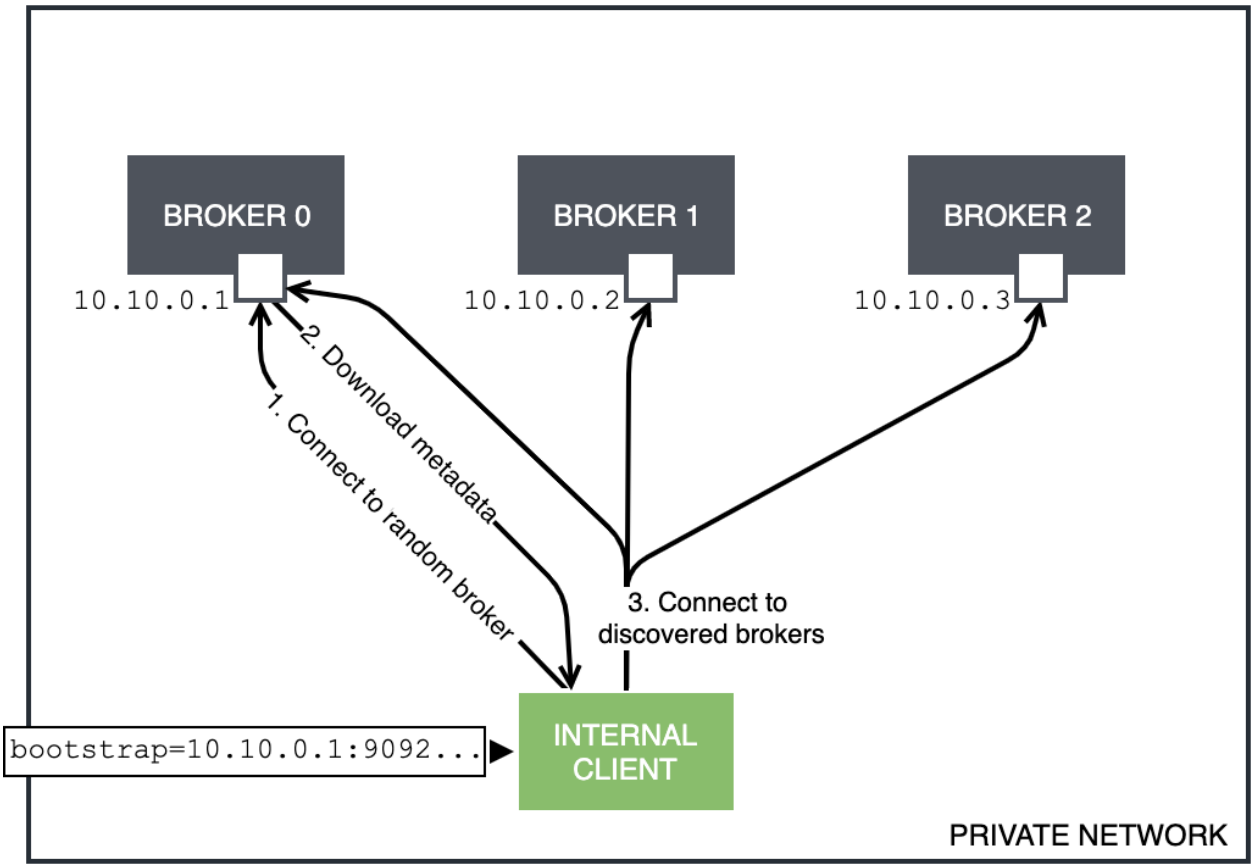
In addition to specifying a socket listener in `listeners`, you need to state how the listener is advertised to producer and consumer clients. This is done by appending an entry to `advertised.listeners`, in the form of `<listener protocol>://<advertised host name>:<advertised port>`. Returning to our earlier example, we would like the first broker to be advertised on `10.10.0.1:9092`. So we would edit `server.properties` to the tune of:

```
advertised.listeners=PLAINTEXT://10.10.0.1:9092
```

Note: There was no need to change `listeners` or `listener.security.protocol.map` because we didn't introduce a new listener; we simply changed how the existing listener is advertised. In a multi-broker setup, we would make similar changes to the other brokers.

Don't forget to restart Kafka after changing any values in `server.properties`.

How does this fix bootstrapping? The client will still connect to a random host specified in the bootstrap list. This time, the class metadata returned by the host will contain the correct client-reachable addresses and port numbers of broker nodes, rather than a set of `localhost:9092` entries. Now the client is able to establish direct connections, provided that these addresses are reachable from the client. The diagram below illustrates this.



Bootstrapping over an internal network

Kafdrop comes in useful for understanding the topology of the cluster and gives some clues as to the listener configuration. The cluster overview screen shows both the bootstrap list, as well as the address and port reported in the cluster metadata.

Kafdrop, showing broker metadata

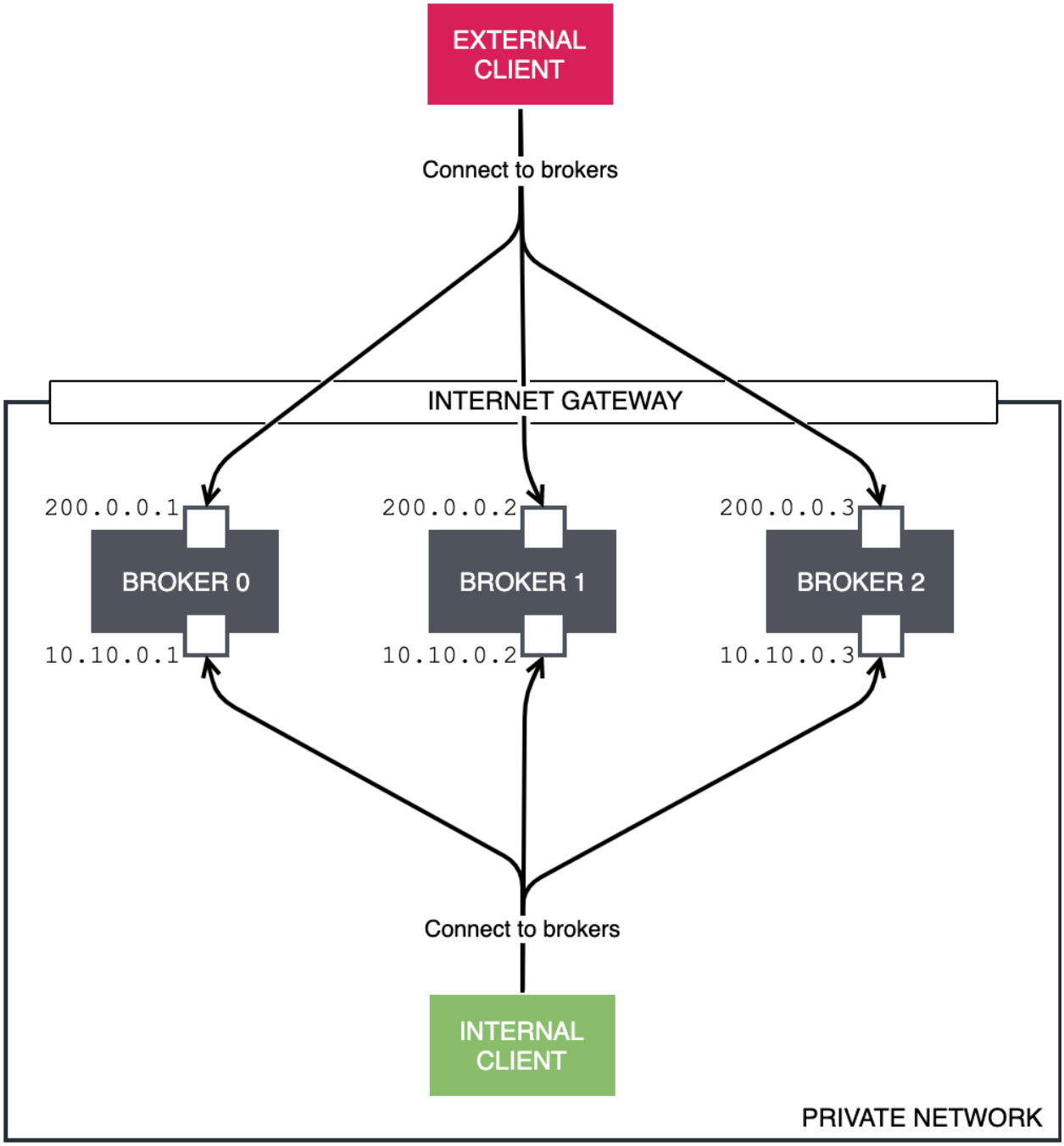
Earlier in this chapter, we touched upon the `client.dns.lookup` property and the effect of setting it to `use_all_dns_ips` — specifically, the Java client's ability to utilise all resolved DNS entries. When enabled, the advertised listeners will also be subjected to the same DNS treatment. If an advertised listener resolves to multiple IP addresses, the client will cycle through these until a connection is established.

Multiple listeners

The simple example discussed earlier applies when there is a single ingress point into the Kafka cluster; every client, irrespective of their type or deployment location, accesses the cluster via that ingress.

What if you had multiple ingress points? Suppose our three-broker cluster is deployed in a virtual private cloud (VPC) on AWS (or some other Cloud). Most clients are also deployed within the same VPC. However, a handful of legacy consumer and producer applications are deployed outside the VPC in a managed data centre. There are no private links (VPN or Direct Connect) between the VPC and the data centre.

One approach is to expose the brokers to the outside world via an Internet Gateway, such that each broker has a pair of addresses — an internal address and an external address. Assume that security is a non-issue for the moment — we don't care about encryption, authentication or authorization — we just want to connect to the Kafka cluster over the Internet. The internal addresses will be lifted from the last example, while the external ones will be `200.0.0.1`, `200.0.0.2`, and `200.0.0.3`. The desired broker and client topology is illustrated below.



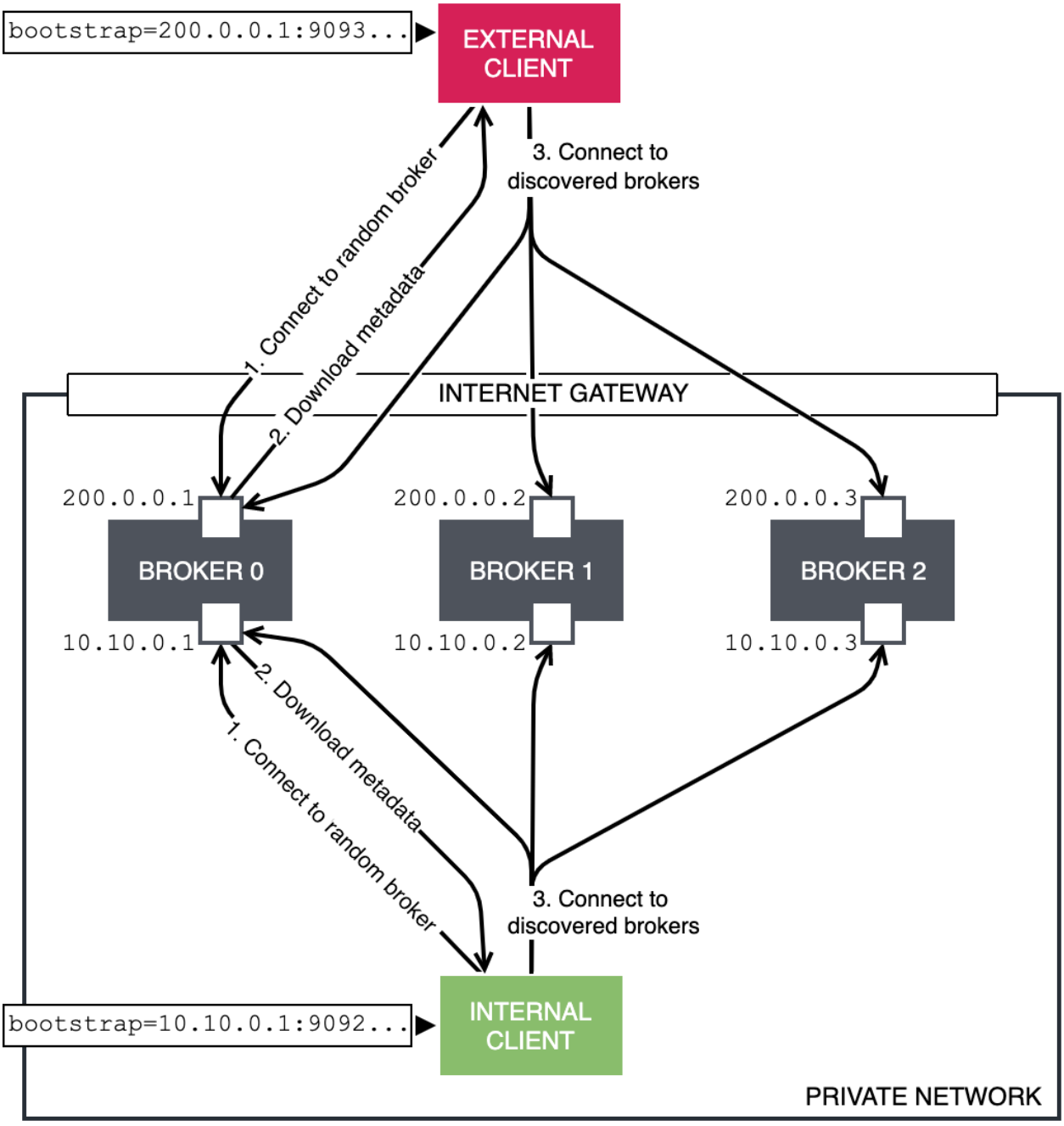
Connecting from two locations

As you would have guessed by now, setting the bootstrap list to `200.0.0.1:9092,200.0.0.2:9092,200.0.0.3:9092` will not work for external clients. The client will make the initial connection, download the cluster metadata, then attempt fruitlessly to connect to one of the `10.10.0.x` private addresses. Brokers need a way of distinguishing internal clients from external clients so that a tailored set of cluster metadata can be served depending on where the client is connecting from.

The situation is resolved by adding a second listener, targeting the external ingress. We would have to modify our `server.properties` to resemble the following:

```
listeners=INTERNAL://:9092,EXTERNAL://:9093
advertised.listeners=INTERNAL://10.10.0.1:9092,\
    EXTERNAL://200.0.0.1:9093
listener.security.protocol.map=INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT
inter.broker.listener.name=INTERNAL
```

Rather than calling our second listener `PLAINTEXT2`, we've gone with something sensible — the listener protocols were named `INTERNAL` and `EXTERNAL`. The `advertised.listeners` property is used to segregate the metadata based on the specific listener that handled the initial bootstrapping connection from the client. In other words, if the client connected on the `INTERNAL` listener socket bound on port `9092`, then the cluster metadata would contain `10.10.0.1:9092` for the responding broker as well as the corresponding `INTERNAL` advertised listener addresses for its peer brokers. Conversely, if a client was bootstrapped to the `EXTERNAL` listener socket on port `9093`, then the `EXTERNAL` advertised addresses are served in the metadata. The illustration below puts it all together.



Bootstrapping from two locations

Individual listener configuration for every Kafka broker node is persisted centrally in the ZooKeeper cluster and is perceived identically by all Kafka brokers. Naturally, this implies that **brokers must be configured with identical listener names**; otherwise, each broker will serve different cluster metadata to their clients.

In addition to the changes to `listeners` and `advertised.listeners`, corresponding entries are also required in the `listener.security.protocol.map`. Both the `INTERNAL` and `EXTERNAL` listener protocol

names have been mapped to the `PLAINTEXT` security protocol.

Keeping with the tradition of dissecting one Kafka feature at a time, we have gone with the simplest `PLAINTEXT` connections in this example. From a security standpoint, this is clearly not the approach one should take for a production cluster. Security protocols, authentication, and authorization will be discussed in Chapter 16: Security.

Clients are not the only applications connecting to Kafka brokers. Broker nodes also form a mesh network, connecting to one another to satisfy internal replication objectives — ensuring that writes to partition leaders are reflected in the follower replicas. The internal mesh connections are referred to as *inter-broker communications*, and use the same wire protocol and listeners that are exposed to clients. Since we changed the internal listener protocol name from the default `PLAINTEXT` to `INTERNAL`, we had to make a corresponding change to the `inter.broker.listener.name` property. This property does not appear in the out-of-the-box `server.properties` file — it must be added manually.

A port may not be bound to by more than one listener on the same network interface. As such, *the port number must be unique for any given interface*. If leaving the interface address unspecified, or if providing a `0.0.0.0` meta-address, one must assign a unique port number for each listener protocol. In our example, we went with `9092` for the internal and `9093` for the external route.

> The discussion on ports and network interfaces leads to a minor curiosity: What if we attempted to assign the same port number to both listeners by explicitly qualifying the IP address? Would this even work?
>
> ```
> listeners=INTERNAL://10.10.0.1:9092,EXTERNAL://200.0.0.1:9092
> ```
>
> The answer depends on whether the host has (physical or virtual) network interfaces that correspond to these IP addresses. Most hosts will have at least two effective addresses available for binding: `localhost` and the address assigned to an Ethernet interface. Some hosts will have more addresses — due to additional network interfaces, IPv6, and virtual interface drivers.
>
> In some configurations, the host's provisioned IP addresses may be assigned directly to network interfaces. For example, dual-homed hosts will typically have a dedicated network interface for each address. Specifying the routable address in the listener configuration will work as expected — the listener's backing server socket will be bound to the correct network interface, avoiding a port conflict. However, when using a NAT (Network Address Translation) device such as an Internet Gateway to selectively assign public IP addresses to hosts that are on an otherwise private network, the host will have no knowledge of its public IP address. Furthermore, it might only have one network interface attached. In these cases, specifying the host's routable address in the `listeners` property will cause a port conflict.

You might be wondering: How often do I need to deal with advertised listeners? Most non-trivial broker configurations support multiple ingress points, often more than two. Most vendors of man-

aged Kafka clusters that offer peering arrangements with public cloud providers will also give you the option of connecting from multiple networks (peered and non-peered). To top it off, there is another common example of multiple listeners coming up.

Listeners and the Docker Network

These days it's common to see a complete application stack deployed across several Docker containers linked by a common network. Starting with local testing, tools like *Docker Compose* make it easy to wire up a self-contained application stack, comprising back-end services, client-facing APIs, databases, message brokers, and so on — spun up rapidly on a developer's machine, then torn down when not needed. Taking it up a notch, orchestration platforms like *Kubernetes*, *OpenShift*, *Docker Swarm*, and *AWS Elastic Container Services* add auto-scaling, zero-downtime deployments, and service discovery into the mix, for a production-grade containerised deployment.

A solid understanding of Kafka's listener and client bootstrapping mechanism is essential to deploying a broker in a containerised environment. The upcoming example will illustrate the use of multiple listeners in a basic application stack, comprising ZooKeeper, Kafka, and Kafdrop. Docker Compose will bind everything together, so some knowledge of Compose is assumed. To spice things up, we will expose Kafka outside of the Compose stack.

Before you run this example, ensure that Kafka, ZooKeeper, and Kafdrop instances that you may have running from previous exercises have been stopped.

To get started, create a `docker-compose.yaml` file in a directory of your choice, containing the following snippet:

```
version: "3.2"
services:
  zookeeper:
    image: bitnami/zookeeper:3
    ports:
      - 2181:2181
    environment:
      ALLOW_ANONYMOUS_LOGIN: "yes"
  kafka:
    image: bitnami/kafka:2
    ports:
      - 9092:9092
    environment:
      KAFKA_CFG_ZOOKEEPER_CONNECT: zookeeper:2181
      ALLOW_PLAINTEXT_LISTENER: "yes"
```

```
      KAFKA_LISTENERS: >-
          INTERNAL://:29092,EXTERNAL://:9092
      KAFKA_ADVERTISED_LISTENERS: >-
          INTERNAL://kafka:29092,EXTERNAL://localhost:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: >-
          INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: "INTERNAL"
    depends_on:
      - zookeeper
  kafdrop:
    image: obsidiandynamics/kafdrop:latest
    ports:
      - 9000:9000
    environment:
      KAFKA_BROKERCONNECT: kafka:29092
    depends_on:
      - kafka
```

Then bring up the stack by running `docker-compose up` in a terminal. (This must be run from the same directory where the `docker-compose.yaml` resides.) Once it boots, navigate to localhost:9000[12] in your browser. You should see the Kafdrop landing screen.

It's the same Kafdrop application as in the previous examples; the only minor difference is the value of the 'Bootstrap servers'. In this example, we are bootstrapping Kafdrop using `kafka:29092` — being the internal ingress point to the Kafka broker. The term 'internal' here refers to all network traffic originating from within the Compose stack. Containers attached to the Docker network are addressed simply by their *service name*, while the mechanics of Docker Compose by default prevent the traffic from leaving the Docker network.

Externally (that is, outside of Docker Compose), we can access both Kafka and Kafdrop with the aid of the port bindings defined in `docker-compose.yaml` file. Let's find out if this actually works. Create a test topic using the Kafka CLI tools:

```
$KAFKA_HOME/bin/kafka-topics.sh \
    --bootstrap-server localhost:9092 \
    --create --topic test \
    --replication-factor 1 \
    --partitions 4
```

Now try listing the topics:

[12] http://localhost:9000

```
$KAFKA_HOME/bin/kafka-topics.sh \
    --bootstrap-server localhost:9092 \
    --list
```

You should see a single entry echoed to the terminal:

```
test
```

Switch back to your browser and refresh Kafdrop. As expected, the `test` topic appears in the list.

Dissecting the `docker-compose.yaml` file, we set up three services. The first is `zookeeper`, launched using the `bitnami/zookeeper` image. This is the first container to start, as it has no declared dependencies. The next service is `kafka`, which declares its dependence on the `zookeeper` service. Finally, `kafdrop` declares its dependence on `kafka`.

The `kafka` service presents the most elaborate configuration of the three. The Bitnami Kafka image allows the user to override values in `server.properties` by passing environment variables. This configuration should be familiar to the reader — it is effectively identical to our earlier example, where traffic was segregated using the `INTERNAL` and `EXTERNAL` listener protocol names.

Dependencies in Docker Compose must form a directed acyclic graph. Simply declare the dependencies of each service; Docker Compose will start the services in the correct order, provided no circularities exist. It is a good practice to structure `docker-compose.yaml` in reverse-dependency order — putting the providers towards the beginning of the `docker-compose.yaml`, so that it roughly aligns with how the services will actually be launched.

You can bring down the stack once you are done. Press `CTRL+C` in the terminal window running Docker Compose. This might take a few seconds, as Compose will attempt to shut down the stack gracefully. Once the prompt reappears, destroy the stack by running `docker-compose down -v`. This will also have the effect of deleting any named or anonymous volumes attached to containers.

Bootstrapping is a complex, multi-stage process that enables clients to discover and maintain connections to all brokers in a Kafka cluster. It is complex not just in its internal workings; there really is a fair amount of configuration one must come to terms with in order to comfortably operate a single Kafka instance or a multi-node cluster, whether it be exposed on one or multiple ingress points.

This chapter has taken us through the internal mechanics of bootstrapping. We came to appreciate the design limitations of Kafka's publishing protocol and how this impacts the client-broker relationship, namely, requiring every client to maintain a dedicated connection to every broker in

the cluster. We established how clients engage brokers in directory-style address lookups, using cluster metadata to learn the broker topology and adapt to changes in the broker population. Various traffic segregation scenarios were discussed, exploring the effects of the `listeners` and `advertised.listeners` configuration properties on how cluster metadata is crafted and served to the clients. The `listeners` property configures the sockets that a broker will listen on for incoming connections, while `advertised.listeners` guides the clients in resolving the brokers. Finally, the use of Docker Compose served a practical everyday example of how listener-based segregation is employed on a routine basis to run Kafka in a containerised environment.

Chapter 9: Broker Configuration

Reminiscing on our Kafka journey, so far we have *mostly* gotten away with running a fairly 'vanilla' broker setup. The exception, of course, being the tweaks to the `listeners` and `advertised.listeners` configuration properties that were explored in the course of Chapter 8: Bootstrapping and Advertised Listeners.

As one might imagine, Kafka offers a myriad of configuration options that affect various aspects of its behaviour, ranging from the substantial to the minute. The purpose of this chapter is to familiarise the reader with the *core* configuration concepts, sufficient to make one comfortable in making changes to all aspects of Kafka's behaviour, using a combination of static and dynamic mechanisms, covering a broad set of configurable entities, as well as targeted canary-style updates.

The knowledge gained in this chapter should be complemented with the official Apache Kafka online reference documentation available at kafka.apache.org/documentation[13]. There are literally hundreds of configurable elements described in rigorous detail in the reference documentation. Furthermore, each major release of Kafka often brings about new configuration and deprecates some of the existing. There is little value in regurgitating online reference material in a published book; instead, this chapter will focus on the fundamentals, which are incidentally least catered to by the online documentation.

Entity types

There are four *entity types* that a configuration entry may apply to:

1. `brokers`: One or more Kafka brokers.
2. `topics`: Existing or future topics.
3. `clients`: Producer and consumer clients.
4. `users`: Authenticated user principals.

The `brokers` entity type can be configured either statically — by amending `server.properties`, or dynamically — via the Kafka Admin API. Other entity types may only be administered dynamically.

Dynamic update modes

Orthogonal to the entity types, there are three *dynamic update modes* of configuration entries that affect broker behaviour:

[13] http://kafka.apache.org/documentation

1. `read-only`: The configuration is effectively static, requiring a change to `server.properties` and a subsequent broker restart to come into effect.
2. `per-broker`: May be updated dynamically for each broker. The configuration is applied immediately, without requiring a broker restart.
3. `cluster-wide`: May be updated dynamically as a cluster-wide default. May also be updated as a per-broker value for canary testing.

These modes apply to the `brokers` entity type. Each subsequent mode in this list is a strict superset of its predecessors. In other words, properties that support the `cluster-wide` mode, will also support `per-broker` and `read-only` modes. However, supporting `per-broker` does not imply supporting `cluster-wide`. As an example, the `advertised.listeners` property can be defined on a `per-broker` basis, but it cannot be applied `cluster-wide`.

For other entity types, such as per-topic or per-user configuration, the settings can *only* be changed dynamically via the API. The scoping rules are also different. For example, there is no concept of `per-broker` configuration for topics; one can target all topics or individual topics, but the configuration always applies to the entire cluster. (It would make no sense to roll out a topic configuration to just one broker.)

The term *dynamic update mode* might sound a little confusing, particularly for the `read-only` mode. The latter is effectively a static configuration entry defined in the `server.properties` file — it cannot be assigned or updated via the Admin API, and is anything but dynamic.

Dynamic configuration (which excludes the `read-only` mode) is persisted centrally in the ZooKeeper cluster. You don't need to interact with ZooKeeper directly to assign the configuration entries. (This option is still supported for backward compatibility; however, it is deprecated and its use is strongly discouraged.) Instead, the Kafka Admin API and the built-in CLI tools allow an authorized user to impart changes to the Kafka cluster directly, which in turn, will be written back to ZooKeeper.

The 'Broker configs' section in the official Kafka documentation indicates the highest supported dynamic update mode for each configuration entry.

The documentation has a separate 'Topic configs'[14] section, listing configuration properties that target individual topics. There is no dynamic update mode indicated in the documentation; in all cases, the update mode is either per-topic or cluster-wide.

Configuration precedence and defaults

There is a strict precedence order when configuration entries are applied. Stated in the order of priority (the highest being at the top) the precedence chain is made up of:

1. Dynamic per-entity configuration;

[14] http://kafka.apache.org/documentation.html#topicconfigs

2. Dynamic cluster-wide default configuration; followed by
3. Static `read-only` configuration from `server.properties`.

A seasoned Kafka practitioner may have picked up on two additional, easily overlooked, configuration levels which we have not mentioned — *default configuration* and *deprecated configuration*. The default configuration is applied automatically if no other configuration can be resolved. There is a caveat: A property may be a relatively recent addition, replacing an older, deprecated property. If the newer property is not set, Kafka will apply the value of a deprecated property *if one is set*, otherwise, and only then, will the default value be applied. (In fact, a configuration property may shadow multiple deprecated properties, in which case a more complex chain of precedence is formed.)

Configuration defaults are also specified in the official Kafka documentation page, but there is a snag: the default value often does not convey any practical meaning. For example, what does it mean when the default value of `advertised.listeners` is stated as `null`? In addition to viewing the default, one must carefully read the description of the property, particularly when a property replaces one or more deprecated properties. The documentation delineates the behaviour of an entity (broker, topic, etc.) when the property is not set.

Dynamic update modes were introduced in Kafka 1.1 as part of KIP-226[a] (*KIP* stands for Kafka Improvement Proposal) to reduce the overhead of administering a large Kafka cluster and potential for interruptions and performance degradation caused by rolling broker restarts. Some key motivating use cases included —

- Updating short-lived SSL keystores on the broker;
- Performance-tuning based on metrics (e.g. increasing network or I/O threads);
- Adding, removing, or reconfiguring metrics reporters;
- Updating configuration of all topics consistently across the cluster;
- Updating log cleaner configuration for tuning; and
- Updating listener/security configuration.

The original proposal considered deprecating the static configuration entries in `server.properties` altogether for those settings that could be altered programmatically. In the course of analysis and community consultation, this measure was found to be too drastic and consequently rejected — entries in `server.properties` were allowed to stay, as it was considered useful to maintain a simple quick-start experience of working with Kafka without setting up configuration options in ZooKeeper first.

[a] https://cwiki.apache.org/confluence/display/KAFKA/KIP-226+-+Dynamic+Broker+Configuration

Applying broker configuration

Static configuration

The static configuration for all Kafka components is housed in `$KAFKA_HOME/config`. A broker sources its `read-only` static configuration from `server.properties`.

With one exception, all entries in `server.properties` are optional. If omitted, the fallback chain comprising deprecated properties and the default value takes effect. The only mandatory setting is `zookeeper.connect`. It specifies the ZooKeeper connection string as a comma-separated list of `host:port` pairs, in the form `host1:port1,host2:port2,...,hostN:portN` — allowing for multiple redundant ZooKeeper connections.

Other than sharing the comma-separated `host:port` format, the ZooKeeper connection string has nothing in common with the bootstrap servers list used to configure Kafka clients. While a Kafka client employs a two-step process to connect to all brokers in a cluster, a Kafka broker needs only one connection to a ZooKeeper node and will establish it directly. Under the hood, ZooKeeper employs an atomic broadcast protocol to syndicate updates among all peer nodes, eliminating the need for every broker to connect to every node in the ZooKeeper ensemble.

Another crucial configuration entry in `server.properties` is `broker.id`, which specifies the unique identifier of the broker within the cluster. This is being brought up now because awareness of the broker IDs for nodes in a Kafka cluster is required to administer targeted configuration, where changes may be progressively applied to select subsets of the broker nodes.

If left unspecified or set to `-1`, the broker ID will be dispensed automatically by the ZooKeeper ensemble, using an atomically-generated sequence starting from `1001`. (This can be configured by setting `reserved.broker.max.id`; the sequence will begin at one higher than this number.)

It is considered good practice to set `broker.id` explicitly, as it makes it easy to quickly determine the ID by looking at `server.properties`. Once the ID has been assigned, it is written to a `meta.properties` file, residing in the Kafka logs directory. This directory is specified by the `log.dirs` property, defaulting to `/tmp/kafka-logs`.

While the naming might appear to suggest otherwise, the Kafka logs directory is not the same as the directory used for application logging. The logs directory houses the log segments, indexes, and checkpoints — used by the broker to persist topic-partition data. The contents of this directory are essential to the operation of the broker; the loss of the logs directory amounts to the loss of data for the broker in question.

To change the ID of an existing broker, you must first stop the broker. Then remove the `broker.id` entry from `meta.properties` or delete the file altogether. You can then change `broker.id` in

`server.properties`. Finally, restart the broker. Upon startup, the Kafka application log should indicate the new broker ID.

Dynamic configuration

Dynamic configuration is applied remotely over a conventional client connection. This can be done using a CLI tool — `kafka-configs.sh`, or programmatically — using a client library.

The following examples assume that the Kafka CLI tools in `$KAFKA_HOME/bin` have been added to your path. Run `export PATH=$KAFKA_HOME/bin:$PATH` to do this for your current terminal session.

We are going to start by viewing a fairly innocuous configuration entry — the number of I/O threads. Let's see if the value has been set in `server.properties`:

```
grep num.io.threads $KAFKA_HOME/config/server.properties
```

```
num.io.threads=8
```

Right, `num.io.threads` has been set to `8`. This means that our broker maintains a pool of eight threads to manage disk I/O. We can take a peek into the broker process to see these threads:

```
KAFKA_PID=$(jps -l | grep kafka.Kafka | awk '{print $1}')
jstack $KAFKA_PID | grep data-plane-kafka-request-handler
```

```
"data-plane-kafka-request-handler-0" #50 daemon prio=5 os_prio=31 □
    cpu=672.65ms elapsed=18234.06s tid=0x00007faf670dc000 □
    nid=0x13b03 waiting on condition  [0x0000700007836000]
"data-plane-kafka-request-handler-1" #51 daemon prio=5 os_prio=31 □
    cpu=671.82ms elapsed=18234.06s tid=0x00007faf670d9000
    nid=0xc803 waiting on condition  [0x0000700007939000]
"data-plane-kafka-request-handler-2" #52 daemon prio=5 os_prio=31 □
    cpu=672.39ms elapsed=18234.06s tid=0x00007faf670da000 □
    nid=0xca03 waiting on condition  [0x0000700007a3c000]
"data-plane-kafka-request-handler-3" #53 daemon prio=5 os_prio=31 □
    cpu=676.05ms elapsed=18234.06s tid=0x00007faf68802800 □
    nid=0xcc03 waiting on condition  [0x0000700007b3f000]
"data-plane-kafka-request-handler-4" #72 daemon prio=5 os_prio=31 □
    cpu=104.53ms elapsed=16207.48s tid=0x00007faf67b5c800 □
    nid=0x1270b waiting on condition  [0x0000700007f4b000]
```

```
"data-plane-kafka-request-handler-5" #73 daemon prio=5 os_prio=31 □
    cpu=97.75ms elapsed=16207.48s tid=0x00007faf67b5d800 □
    nid=0xe107 waiting on condition  [0x0000700008866000]
"data-plane-kafka-request-handler-6" #74 daemon prio=5 os_prio=31 □
    cpu=96.60ms elapsed=16207.48s tid=0x00007faf650a8800 □
    nid=0x1320b waiting on condition  [0x0000700008969000]
"data-plane-kafka-request-handler-7" #75 daemon prio=5 os_prio=31 □
    cpu=105.47ms elapsed=16207.48s tid=0x00007faf68809000 □
    nid=0x12507 waiting on condition  [0x0000700008a6c000]
```

Indeed, eight threads have been spawned and are ready to handle I/O requests.

Let's use the `--describe` switch to list the dynamic value. The property name passed to `--describe` is optional. If omitted, all configuration entries will be shown.

```
kafka-configs.sh --bootstrap-server localhost:9092 \
    --entity-type brokers --entity-name 0 \
    --describe num.io.threads
```

```
Configs for broker 0 are:
```

It's come up empty. That's because there are no matching `per-broker` dynamic configuration entries persisted in ZooKeeper. Also, this tool does not output static entries in `server.properties` unless they have been overridden by dynamic entries.

A minor note on terminology: this book is aligned with the official Kafka documentation in referring to the 'name' part of the configuration entry as the 'property name'. The CLI tools that ship with Kafka refer to the property name as the 'key'. Working with Kafka, you will eventually become used to seeing the same thing being referred to by vastly different names. Even within the CLI tools, there is little consistency — for example, some tools refer to the list of bootstrap servers with the `--bootstrap-server` parameter, while others use `--broker-list`. To add to the confusion, some tools will accept multiple bootstrap servers as a comma-separated list, but still insist on the singular form of 'bootstrap server' (without the trailing 's'). For all the undisputed merits of Kafka, do try to keep your expectations grounded when working with the built-in tooling.

Let's change the `num.io.threads` value by invoking the following command:

```
kafka-configs.sh --bootstrap-server localhost:9092 \
    --entity-type brokers --entity-name 0 \
    --alter --add-config num.io.threads=4
```

```
Completed updating config for broker: 0.
```

Ru-run the `kafka-configs.sh ... --describe ...` command:

```
Configs for broker 0 are:
  num.io.threads=4 sensitive=false synonyms= □
  {DYNAMIC_BROKER_CONFIG:num.io.threads=4, □
  STATIC_BROKER_CONFIG:num.io.threads=8, □
  DEFAULT_CONFIG:num.io.threads=8}
```

Not only is it now telling us that a `per-broker` entry for `num.io.threads` has been set, but it is also echoing the static value and the default.

View the threads again using the `jstack` command:

```
"data-plane-kafka-request-handler-0" #50 daemon prio=5 os_prio=31 □
    cpu=730.89ms elapsed=18445.03s tid=0x00007faf670dc000 □
    nid=0x13b03 waiting on condition  [0x0000700007836000]
"data-plane-kafka-request-handler-1" #51 daemon prio=5 os_prio=31 □
    cpu=728.40ms elapsed=18445.03s tid=0x00007faf670d9000 □
    nid=0xc803 waiting on condition  [0x0000700007939000]
"data-plane-kafka-request-handler-2" #52 daemon prio=5 os_prio=31 □
    cpu=729.16ms elapsed=18445.03s tid=0x00007faf670da000 □
    nid=0xca03 waiting on condition  [0x0000700007a3c000]
"data-plane-kafka-request-handler-3" #53 daemon prio=5 os_prio=31 □
    cpu=737.62ms elapsed=18445.03s tid=0x00007faf68802800 □
    nid=0xcc03 waiting on condition  [0x0000700007b3f000]
```

Bingo! The number of threads has been reduced, with the action taking effect almost immediately following the update of the dynamic configuration property.

Dynamic configuration is an impressively powerful but dangerous tool. A bad value can take an entire broker offline or cause it to become unresponsive. As a precautionary measure, Kafka caps changes of certain numeric values to either half or double their previous value, forcibly smoothening out changes in the configuration. Increasing a setting to over double its initial value or decreasing it to under half of its initial value requires multiple incremental changes.

Where a setting supports `cluster-wide` scoping, a good practice is to apply the setting to an individual broker before propagating it to the whole cluster. This is effectively what we've done with `num.io.threads`.

The `--entity-type` argument is compulsory and can take the value of `brokers`, `users`, `clients`, or `topics`. The `--entity-name` argument is compulsory for `per-broker` dynamic updates and can be replaced with `--entity-default` for `cluster-wide` updates. Let's apply the `num.io.threads.setting` to the entire cluster.

```
kafka-configs.sh --bootstrap-server localhost:9092 \
    --entity-type brokers --entity-default \
    --alter --add-config num.io.threads=4
```

Then list the configuration:

```
kafka-configs.sh --bootstrap-server localhost:9092 \
    --entity-type brokers --entity-name 0 \
    --describe num.io.threads
```

```
Configs for broker 0 are:
  num.io.threads=4 sensitive=false synonyms= □
  {DYNAMIC_BROKER_CONFIG:num.io.threads=4, □
  DYNAMIC_DEFAULT_BROKER_CONFIG:num.io.threads=4, □
  STATIC_BROKER_CONFIG:num.io.threads=8, □
  DEFAULT_CONFIG:num.io.threads=8}
```

The resulting list has an extra value: `DYNAMIC_DEFAULT_BROKER_CONFIG:num.io.threads=4`. As previously mentioned, the `cluster-wide` entries are second in the order of precedence, which is also reflected in the list above. When using the `--describe` switch, we can specify `--entity-default` to isolate `cluster-wide` configuration.

```
kafka-configs.sh --bootstrap-server localhost:9092 \
    --entity-type brokers --entity-default \
    --describe num.io.threads
```

```
Default config for brokers in the cluster are:
  num.io.threads=4 sensitive=false synonyms= □
  {DYNAMIC_DEFAULT_BROKER_CONFIG:num.io.threads=4}
```

Having applied the `cluster-wide` setting, and assuming the update has proved to be stable, we can now remove the `per-broker` entry. Run the following:

```
kafka-configs.sh --bootstrap-server localhost:9092 \
    --entity-type brokers --entity-name 0 \
    --alter --delete-config num.io.threads
```

Always remove `per-broker` settings once you apply the `cluster-wide` defaults. Unless there is a compelling reason for a `per-broker` setting to vary from a `cluster-wide` default, maintaining both settings creates clutter and may lead to confusion in the longer-term — when the `cluster-wide` setting is altered later down the track.

Having just deleted the `per-broker` configuration, presumably the only remaining entry is the `cluster-wide` one. Let's describe the configuration for broker `0`:

```
kafka-configs.sh --bootstrap-server localhost:9092 \
    --entity-type brokers --entity-name 0 \
    --describe num.io.threads


Configs for broker 0 are:
```

This may leave the reader in a slightly puzzled state. Where did the `DYNAMIC_DEFAULT_BROKER_CONFIG` entry go? When invoked with the `--entity-name` parameter, `kafka-configs.sh` will only display an entry if a `per-broker` configuration has been assigned, irrespective of whether or not a `cluster-wide` entry has also been set. Running the command with the `--entity-default` switch still works as expected, showing the `cluster-wide` defaults:

```
Default config for brokers in the cluster are:
  num.io.threads=4 sensitive=false synonyms= □
  {DYNAMIC_DEFAULT_BROKER_CONFIG:num.io.threads=4}
```

Now that we're done with our examples, we can revert the configuration to its original state by deleting the `cluster-wide` entry:

```
kafka-configs.sh --bootstrap-server localhost:9092 \
    --entity-type brokers --entity-default \
    --describe num.io.threads
```

Applying topic configuration

Settings that apply broadly to all topics, as well as topic-wide defaults, can be edited using the static configuration in `server.properties` or via dynamic updates. In addition, some settings can be altered on a per-topic basis using just the dynamic approach.

The next example will tinker with another fairly benign setting — `flush.messages` — which controls the number of messages that can be written to a log before it is forcibly flushed to disk with the `fsync` command. Log flushing is disabled by default. (Actually, the value is set to a very high number: $2^{63} - 1$.)

Kafka requires that the topic exists before making targeted changes. We are going to create a test topic named `test.topic.config` for this demonstration. Run the command below.

```
kafka-topics.sh --bootstrap-server localhost:9092 --create \
    --topic test.topic.config --replication-factor 1 --partitions 1
```

The next command will apply an override for `flush.messages` for the `test.topic.config` topic.

```
kafka-configs.sh --zookeeper localhost:2181 \
    --entity-type topics --entity-name test.topic.config \
    --alter --add-config flush.messages=100
```

Note: When referring to the `topics` entity type, you must substitute the `--bootstrap-server` argument with `--zookeeper`, specifying the `host:port` combination of any node in the ZooKeeper ensemble.

This is not a limitation of Kafka — the standard Admin API supports setting and querying topic-level configuration; rather, the limitation is with the CLI tools that seem to have taken a back seat to the rapid improvements in the Kafka wire protocol and client APIs. This limitation applies to Kafka 2.4.0 and earlier versions, verified at the time of writing. For those interested, a solution proposal is being floated in KIP-248[15]. There is a sliver of hope that by the time this book falls into the reader's hands, the maintainers of Kafka will have addressed it.

We can now use the `--describe` switch to read back the configuration:

```
kafka-configs.sh --zookeeper localhost:2181 \
    --entity-type topics --entity-name test.topic.config \
    --describe flush.messages


Configs for topic 'test.topic.config' are flush.messages=100
```

It was previously stated that the entity name is an optional argument to the `--describe` switch. To further broaden the query to *all* topics, use the `--entity-default` switch, as shown below.

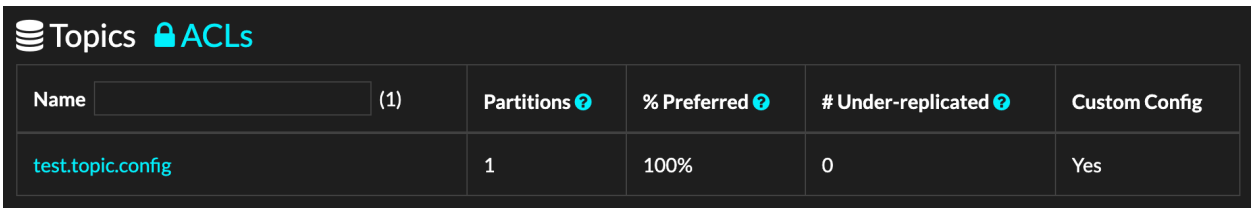
```
kafka-configs.sh --zookeeper localhost:2181 \
    --entity-type topics --entity-default --describe
```

Similarly, using `--entity-default` with the `--alter` switch will apply the dynamic configuration defaults to *all* topics. For example, if we wanted to apply `flush.messages=100` to all topics, we could have run the following:

```
kafka-configs.sh --zookeeper localhost:2181 \
    --entity-type topics --entity-default \
    --alter --add-config flush.messages=100
```

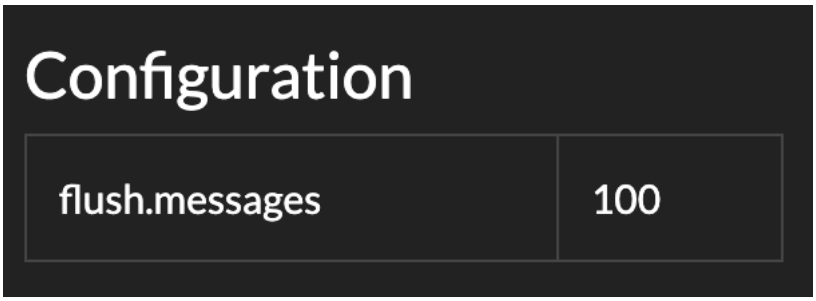
[15] https://cwiki.apache.org/confluence/display/KAFKA/KIP-248+-+Create+New+ConfigCommand+That+Uses+The+New+AdminClient

Using `kafka-configs.sh` is one way of viewing the topic configuration; however, the limitation of requiring ZooKeeper for topic-related configuration prevents its use in most production environments. There are other alternatives — for example, Kafdrop — that display topic configuration using the standard Kafka Admin API. Switch to Kafdrop and refresh the cluster overview page. You should see the recently-created `test.topic.config` topic appear in the topics list. Furthermore, there is an indication that a custom configuration has been assigned.



Kafdrop cluster overview, showing topic names

Click through to the topic. The custom configuration is displayed in the top-right corner of the topic overview screen.



Kafdrop topic overview, showing custom configuration

Having done with this example, we can revert the configuration to its original state by running the command below.

```
kafka-configs.sh --zookeeper localhost:2181 \
    --entity-type topics --entity-name test.topic.config \
    --alter --delete-config flush.messages
```

Users and Clients

The lion's share of configuration use cases relates to brokers, with the remainder mostly falling on topics. Users and clients are configured within the broader context of security and quota management — topic areas that will be covered separately in Chapter 16: Security and Chapter 17: Quotas.

Although users and clients are not going to be covered in this chapter, their configuration closely resembles that of topics. In other words, they support two dynamic update modes: per-entity and

cluster-wide, with the per-entity taking precedence, followed by cluster-wide defaults, and finally by the hard defaults.

Kafka is a highly tunable and adaptable event streaming platform. Understanding the ins and outs of Kafka configuration is essential to operating a production cluster at scale.

Kafka provides a fair degree of scope granularity of an individual configuration entry. Configuration entries may apply to a range of entity types, including brokers, topics, users, and clients. Depending on the nature of the configuration, a change may be administered either statically — by amending `server.properties`, or dynamically — via the Kafka Admin API. A combination of whether remote changes are supported and the targeted scope of a configuration entry is referred to as its *dynamic update mode*. The broadest of supported dynamic update modes applies a cluster-wide setting to all entities of a given type. The per-entity dynamic mode allows the operator to target an individual entity — a specific broker, user, topic, or client. The read-only dynamic update mode bears the narrowest of scopes, and is reserved for static, per-broker configuration.

In addition to navigating the theory, we have also racked up some hands-on time with the `kafka-configs.sh` built-in CLI tool. This utility can be used to view and assign dynamic configuration entries. And while it is effective, `kafka-configs.sh` has its idiosyncrasies and limitations — for example, it requires a direct ZooKeeper connection for administering certain entity types.

Chapter 10: Client Configuration

Some of the previous chapters have gotten us well underway in publishing and consuming records, without dwelling on the individual client properties. This is a good time to take a brief detour and explore the world of client configuration.

In contrast to broker configuration, with its dynamic update modes, selective updates, and baffling CLI tools, configuring a client is *comparatively* straightforward — in that client configuration is static and *mostly* applies to the instance being configured. (There are a few exceptions.) However, client configuration is significantly more nuanced — requiring a greater degree of insight on the user's behalf.

This chapter subdivides the configuration into producer, consumer, and admin client settings. Some configurable items are common, and have been extracted into their own section accordingly. The reader may consult this chapter in addition to the online documentation, available at kafka.apache.org/documentation[16]. However, the analysis presented here is much more in-depth, covering material that is not readily available from official sources.

Of the numerous client configuration properties, there are several dozen that relate to performance tuning and various esoteric connection-related behaviour. The primary focus of this chapter is to cover properties that either affect the client functionally, or qualitatively impact any of the guarantees that might be taken for granted. As a secondary objective, the chapter will outline the performance implications of the configuration where the impacts are material or at least perceptible, but the intention is not to cover performance in fine detail — the user may want to defer to more specialised texts for a finer level of analysis. Other chapters will also cover aspects of Kafka's performance in more detail.

Configuration gotchas

Client configuration is arguably more crucial than broker configuration. Broker configuration is administered by teams or individuals who are responsible for the day-to-day operation of Kafka, likely among a few other large items of infrastructure under their custodianship. Changes at this level are known or at least assumed to be broadly impacting, and are usually made with caution, with due notice given to end-users. Furthermore, the industry is seeing a noticeable shift towards fully-managed Kafka offerings, where broker operation falls under the custodianship of expert teams. Managing a single technology for a large number of clients eventually makes one proficient at it.

By comparison, there really is no such thing as a 'fully-managed Kafka client'. Clients are bespoke applications targeting specific business use cases and written by experts in the application domain.

[16] https://kafka.apache.org/documentation

We are talking about software engineers with T-shaped skills, who are adept at a broad range of technologies and likely specialising in a few areas — probably closer to software than infrastructure. Personal experience working with engineering teams across several clients in different industries has left a lasting impression upon the author. The overwhelming majority of perceived issues with Kafka are caused by the misuse of the technology rather than the misconfiguration of the brokers. To that point, a typical Kafka end-user knows little more than what *they* consider to be the minimally essential amount of insight they need for their specific uses of Kafka. And herein lies the problem: How does one assess what is essential and what is not, if they don't invest the time in exploring the breadth of knowledge at their disposal?

Software engineers working with Kafka will invariably grasp some key concepts of event streaming. A lot of that insight is amassed from conversations with colleagues, Internet articles, perusing the official documentation, and even observing how Kafka responds to different settings. Some of it may be speculative, incomplete, assumed, and in more dire cases, outright misleading. Here is an example. Most users know that *Kafka offers strong durability guarantees with respect to published records*. This statement is made liberally by the project's maintainers and commercial entities that derive income from Apache Kafka. This is a marketing phrase; and while it is not entirely incorrect, it has no tangible meaning and can be contorted to imply whatever the user wants it to. Can it be taken that —

1. Kafka never loses a record?
2. Kafka may occasionally lose a record, where 'occasionally' implies a tolerable level? (In which case, who decides on what is tolerable and what is not?)
3. This guarantee is applied by default to all clients and topics?
4. Kafka offers this guarantee as an option but it is up to the client to explicitly take advantage of it?

The answer is a mixture of *#2* and *#4*, but it is much more complex than that. Granted, the cluster will set a theoretical upper bound on the durability metric, which is influenced by the number of brokers and some notional recovery point objective attributed to each broker. In other words, the configuration, topology, and hardware specification of brokers sets the absolute *best-case* durability rating. But it is ultimately the producer client that controls the durability of records both at the point of topic creation — by specifying the replication factor and certain other parameters, and at the point of publishing the record — by specifying the number of acknowledgments for the partition leader to request *and* also waiting for the receipt of the acknowledgment from the leader before deeming the record as published. Most users will take comfort in knowing they have a solid broker setup, neglecting to take the due actions on the client-side to ensure *end-to-end* durability.

The importance of client configuration is further underlined by yet another factor, one that stems from the design of Kafka. Given the present-day claims around the strengths of Kafka's ordering and delivery guarantees, one would be forgiven for assuming that the configuration defaults are sensible, insofar as they ought to favour safety over other competing qualities. In reality, that is not the case. Historically, Kafka was born out of LinkedIn's need to move a very large number of messages efficiently, amounting to multiple terabytes of data on an hourly basis. The loss of a

message was not deemed as catastrophic, after all, a message or post on LinkedIn is hardly more than an episode of self-flattery. Naturally, this has reflected on the philosophy of setting default values that prioritise performance over just about everything else that counts. This proverbial snake-laden pit has bested many an unsuspecting engineer.

When working with Kafka, remember the first rule of optimisation: Don't do it. In fairness, this rule speaks to *premature* optimisation; however, as it happens, most optimisation in Kafka is premature. The good news is: Setting the configuration properties to warrant safety has only a minor impact on performance — Kafka is still a performance powerhouse.

As we explore client configuration throughout the rest of the chapter, pay particular attention to callouts that underline safety. There are a fair few, and each will have a cardinal impact on your experience with Kafka. This isn't to say that configuration 'gotchas' are exclusive to the client-side; broker configuration has them too. Comparatively, the client configuration has a disproportionate amount.

Applying client configuration

Client configuration is assembled as a set of key-value pairs before instantiating a `KafkaProducer`, `KafkaConsumer`, or a `KafkaAdminClient` object. The original way of assembling client properties, dating to the earliest release of Kafka, was to use a `Properties` object:

```
var props = new Properties();
props.setProperty("bootstrap.servers",
                  "localhost:9092");
props.setProperty("key.serializer",
                  StringSerializer.class.getName());
props.setProperty("value.serializer",
                  StringSerializer.class.getName());
props.setProperty("max.in.flight.requests.per.connection",
                  String.valueOf(1));

try (var producer = new KafkaProducer<>(props)) {
  // do something with producer
}
```

A minor annoyance of `Properties`-based configuration is that it forces you to use a `String` type for both keys and values. This makes sense for keys, but values should just be derived from their object representation. Over time, Kafka clients have been enhanced to accept an instance of `Map<String, Object>`. Things have moved on a bit, and the same can now be written in a slightly more succinct way:

```
Map<String, Object> config =
    Map.of("bootstrap.servers", "localhost:9092",
           "key.serializer", StringSerializer.class.getName(),
           "value.serializer", StringSerializer.class.getName(),
           "max.in.flight.requests.per.connection", 1);

try (var producer = new KafkaProducer<>(config)) {
  // do something with producer
}
```

Frankly, whether you use `Properties` or a `Map` has no material bearing on the outcome, and is a matter of style. `Map` offers a terser syntax, particularly when using a Java 9-style `Map.of(...)` static factory method, and has the additional benefit of creating an immutable map. It is also considered the more 'modern' approach by many practitioners. On the flip side, `Properties` forces you to acknowledge that the value is a string and perform type conversion manually. The `Properties` class also has a convenient `load(Reader)` method for loading a `.properties` file. Most of the code in existence that uses Kafka still relies on `Properties`.

When a client is instantiated, it verifies that the keys correspond to valid configuration property names that are supported in the context of that client type. Failing to meet this requirement will result in a warning message being emitted via the configured logger. Let's instantiate a producer client, intentionally misspelling one of the property names:

```
16:49:51/0     WARN  [main]: The configuration □
    'max.in.flight.requests.per.connectionx' was supplied but □
    isn't a known config.
16:49:51/4     INFO  [main]: Kafka version: 2.4.0
16:49:51/5     INFO  [main]: Kafka commitId: 77a89fcf8d7fa018
16:49:51/5     INFO  [main]: Kafka startTimeMs: 1576648191386
```

Kafka developers have opted for a failsafe approach to handling property names. Despite failing the test, the client will continue to operate using the remaining properties. In other words, there is no exception — just a warning log. The onus is on the user to inspect the configuration for correctness and sift through the application logs.

Rather than supplying an unknown property name, let's instead change the value to an unsupported type. Our original example had `max.in.flight.requests.per.connection` set to `1`. Changing the value to `foo` produces a runtime exception:

```
Exception in thread "main" org.apache.kafka.common.config.
    ConfigException: Invalid value foo for configuration
    max.in.flight.requests.per.connection: Not a number of type INT
  at org.apache.kafka.common.config.ConfigDef.parseType
      (ConfigDef.java:726)
  at org.apache.kafka.common.config.ConfigDef.parseValue
      (ConfigDef.java:474)
  at org.apache.kafka.common.config.ConfigDef.parse
      (ConfigDef.java:467)
  at org.apache.kafka.common.config.AbstractConfig.<init>
      (AbstractConfig.java:108)
  at org.apache.kafka.common.config.AbstractConfig.<init>
      (AbstractConfig.java:129)
  at org.apache.kafka.clients.producer.ProducerConfig.<init>
      (ProducerConfig.java:409)
  at org.apache.kafka.clients.producer.KafkaProducer.<init>
      (KafkaProducer.java:326)
  at org.apache.kafka.clients.producer.KafkaProducer.<init>
      (KafkaProducer.java:270)
  at effectivekafka.basic.BasicProducerSample.main
      (BasicProducerSample.java:15)
```

There several gotchas in configuring Kafka clients, and nailing property names is among them. Chapter 11: Robust Configuration explores best-practices for alleviating the inherent naming issue and offers a hand-rolled remedy that largely eliminates the problem.

Common configuration

This section describes configuration properties that are common across all client types, including producers, consumers, and admin clients.

Bootstrap servers

We explored `bootstrap.servers` in Chapter 8: Bootstrapping and Advertised Listeners in considerable detail. The reader is urged to study that chapter, as it provides the foundational knowledge necessary to operate and connect to a Kafka cluster. To summarise, the `bootstrap.server` property is mandatory for all client types. It specifies a comma-delimited list of host-port pairs, in the form `host1:port1,host2:port2,...,hostN:portN`, representing the addresses of a subset of broker nodes that the client can try to connect to, in order to download the complete cluster metadata and subsequently maintain direct connections with *all* broker nodes. The addresses need not all point to

live broker nodes; provided the client is able to reach *at least one* of the brokers, it will readily learn the entire cluster topology.

The crunch is in ensuring that the retrieved cluster metadata lists broker addresses that are reachable from the client. The addresses disclosed in the metadata may be completely different from those supplied in `bootstrap.servers`. As a consequence, the client is able to make the initial bootstrapping connection but stumbles when connecting to the remaining hosts. For a better understanding of this problem and the recommended solutions, consult Chapter 8: Bootstrapping and Advertised Listeners.

Client DNS lookup

The `client.dns.lookup` is a close relative of `bootstrap.servers`, and is also covered in Chapter 8: Bootstrapping and Advertised Listeners. The property is optional, accepting an enumerated constant from the list below.

- `default`: Retains legacy behaviour with respect to DNS resolution, in other words, it will resolve a single address for each bootstrap endpoint — being the first entry returned by the DNS query. This option applies both to the bootstrap list and the advertised hosts disclosed in the cluster metadata.
- `resolve_canonical_bootstrap_servers_only`: Detects aliases in the bootstrap list, expanding them to a list of resolved canonical names using a reverse DNS lookup. This option was introduced in Kafka 2.1.0 as part of KIP-235[17], primarily to support secured connections using Kerberos authentication. This behaviour applies to the bootstrap list only; the advertised hosts are treated conventionally, as per the `default` option.
- `use_all_dns_ips`: Supports multiple `A` DNS records for the same fully-qualified domain name, resolving all hosts for each endpoint in the bootstrap list. The client will try each host in turn until a successful connection is established. This option was introduced in Kafka 2.1.0 as part of KIP-302[18], and applies to both the bootstrap list and the advertised hosts.

Client ID

The optional `client.id` property allows the application to associate a free-form logical identifier with the client connection, used to distinguish between the connected clients. While in most cases it may be safely omitted, the use of the client ID provides for a greater degree of source traceability, as it is used for the logical grouping of requests in Kafka metrics.

Beyond basic traceability, client IDs are also used to enforce quota limits on the brokers. The discussion of this capability will be deferred until Chapter 17: Quotas.

[17] https://cwiki.apache.org/confluence/display/KAFKA/KIP-235%3A+Add+DNS+alias+support+for+secured+connection
[18] https://cwiki.apache.org/confluence/display/KAFKA/KIP-302+-+Enable+Kafka+clients+to+use+all+DNS+resolved+IP+addresses

Retries and retry backoff

The `retries` and `retry.backoff.ms` properties specify the number of retries for transient errors and the interval (in milliseconds) to wait before each subsequent retry attempt, respectively. The number of retries accrues on top of the initial attempt, in other words, the upper bound on the total number of attempts is `retries + 1`.

To clarify, a transient error is any condition that is deemed as *potentially* recoverable. Timeouts are the most common form of transient error, but there are many others that relate to the cluster state — for example, stale metadata or a controller change.

While the `retry.backoff.ms` property applies to all three client types, the `retries` property only exists for the producer and admin clients; it is not supported by the consumer client. Instead of limiting the number of retries, the consumer limits the *total time* accorded to a query — for example, when the `poll()` method is invoked — obviating the need for an explicit retry counter. In spite of this minor disparity, we will discuss these two configuration aspects as a collective whole.

The default setting of `retries` is `Integer.MAX_VALUE`, and the producer and admin clients will wait 100 ms by default between each attempt. There is no default value for the poll timeout that applies to consumer clients — the timeout is specified explicitly as a parameter to the `poll()` method. Whether these defaults are sensible depends on the combination of your network, the amount of resources available to both the cluster and the client apps, and your application's tolerance for awaiting a successful or failed outcome of publishing a record.

There is a gotcha here, albeit a subtle one. It does not fundamentally matter how many retries one permits, or the total time spent retrying, there are only two possible outcomes. The number of retries and the backoff time could be kept to a minimum, in which case the likelihood of an error reaching the application is high. Even if these numbers are empirically derived, eventually one will eventually observe a scenario where the retries are exhausted. Alternatively, one might leave `retries` at its designated default of `Integer.MAX_VALUE`, in which case the client will just keep hammering the broker, while the application fails to make progress. We need to acknowledge that failures are possible and must be accounted for at the application level.

When Kafka was first released into the wild, every broker was self-hosted, and most were running either in a corporate data centre or on a public cloud, in close proximity to the client applications. In other words, the network was rarely the culprit. The landscape has shifted considerably; it is far more common to see managed Kafka offerings, which are delivered either over the public Internet or via VPC peering. Also, with the increased adoption of event streaming in the industry, an average broker now carries more traffic than it used to, with the increase in load easily outstripping the performance advancements attributable to newer hardware and efficiency gains in the Kafka codebase. With the adoption of public cloud providers, organisations are increasingly looking to leverage availability zones to protect themselves from site failures. As a result, Kafka clusters are now larger than ever before, both in terms of the number of broker nodes and their geographic distribution. One needs

to take these factors into account when setting `retries`, `retry.backoff.ms` and the consumer poll timeout, and generally when devising the error handling strategy for the application.

An alternate way of looking at the problem is that it isn't about the stability profile of the underlying network, the capacity of the Kafka cluster, or the distribution of failures one is likely to experience on a typical day. Like any other client consuming a service, one must be aware of their own non-functional requirements and any obligations they might have to their upstream consumers. If a Kafka client application is prepared to wait no more than a set time for a record to be published, then the retry profile and the error handling must be devised with that in mind.

Testing considerations

Continuing the discussion above, another cause of failures that is often overlooked relates to running Kafka in performance-constrained environments as part of automated testing. Developers will routinely run single-broker Kafka clusters in a containerised environment or in a virtual machine. Kafka's I/O performance is significantly diminished in Docker or on a virtualised file system. (The reasons as to why are not important for the moment.) The problem is exacerbated when Docker is run on macOS, which additionally incurs the cost of virtualisation. As a consequence, expect a longer initial readiness time and more anaemic state transitions at the controller. The test may assume that Kafka is available because the container has started and Kafka is accepting connections on its listener ports; however, the latter does not imply that the broker is ready to accept requests. It may take several seconds for it to be ready and the timing will not be consistent from run to run. The default tolerance (effectively indefinite retries) actually copes well with these sorts of scenarios. Tuning retry behaviour to better represent production scenarios, while appearing prudent, may hamper local test automation — leading to brittle tests that occasionally fail due to timing uncertainty.

One way of solving this problem is to allow for configurable retry behaviour, which may legitimately vary between deployment environments. The problem with this approach is it introduces variance between real and test environments, which is rarely ideal from an engineering standpoint. An alternate approach, and one that is preferred by the author, is to introduce an extended wait loop at the beginning of each test, allowing for some grace time for the broker to start up. The loop can poll the broker for some innocuous read-only query, such as listing topic names. The broker may initially take some time to respond, returning a series of errors — which are ignored — while it is still in the process of starting up. But when it does respond, it usually indicates that the cluster has stabilised and the test may commence.

Security configuration

All three client types can be configured for secure connections to the cluster. We are not going to explore security configuration in this chapter, partly because the range of supported options is overbearing, but mostly because this topic area is covered in Chapter 16: Security.

Producer configuration

This section describes configuration options that are specific to the producer client type.

Acknowledgements

The `acks` property stipulates the number of acknowledgements the producer requires the leader to have received before considering a request complete, and before acknowledging the write with the producer. This is fundamental to the durability of records; a misconfigured `acks` property may result in the loss of data while the producer naively assumes that a record has been stably persisted.

Although the property relates to the number of acknowledgements, it accepts an enumerated constant being one of —

- `0`: Don't require an acknowledgement from the leader.
- `1`: Require one acknowledgement from the leader, being the persistence of the record to its local log. This is the default setting when `enable.idempotence` is set to `false`.
- `-1` or `all`: Require the leader to receive acknowledgements from all in-sync replicas. This is the default setting when `enable.idempotence` is set to `true`.

Each of these modes, as well as the interplay between acknowledgements and Kafka's replication protocol, are discussed in detail in Chapter 13: Replication and Acknowledgements.

Maximum in-flight requests per connection

The `max.in.flight.requests.per.connection` property sets an upper bound on the number of unacknowledged requests the producer will send on a single connection before being forced to wait for their acknowledgements. The default value of this property is `5`.

The purpose of this configuration is to increase the throughput of a producer. This is particularly evident over long-haul, high-latency networks, where long acknowledgement times continually interrupt a producer's ability to publish additional records, even if the network capacity otherwise permits this. The problem is not exclusive to high-latency networks; any internal constraint that contributes to increases in acknowledgement times — for example, slow replication within the cluster due to lagging in-sync replicas — will negatively impact the transmission rate.

This is a classic problem of flow control. Anyone familiar with the inner workings of networking protocols will immediately liken the behaviour `max.in.flight.requests.per.connection` to the venerable sliding window protocol used for TCP's flow control. However, it is not quite the same; there is one key distinction — the lack of ordering and reassembly of in-flight records over the extent of the unacknowledged window when idempotence is disabled.

This problem is best explained with an example. Suppose a producer, configured with default values for `max.in.flight.requests.per.connection` and `retries`, queues records *A*, *B*, and *C* to the broker

in that precise order, assuming for simplicity that the records will occupy the same partition. The tacit expectation is that these records will be persisted in the order they were sent, as per Kafka's ordering guarantees. Let's assume that *A* gets to the broker and is acknowledged. A transient error occurs attempting to persist *B*. *C* is processed and acknowledged. The producer, having detected a lack of acknowledgement, will retransmit *B*. Assuming the retransmission is successful, the records will appear in the sequence *A*, *C*, and *B* — distinct from the order they were sent in.

Although the previous example used individual records to illustrate the problem, it was a simplification of Kafka's true behaviour. In reality, Kafka does not forward individual records, but batches of records. But the principle remains essentially the same — just substitute 'record' for 'batch'. So rather than individual arriving out of order, entire batches of records may appear to be reordered on their target partition.

The underlying issue is that the broker implicitly relies on ordering provided by the underlying transport protocol (TCP), which acts at Layer 4 of the OSI model. Being unaware of the application semantics (Layer 7), TCP cannot assist in the reassembly of application-level payloads. By default, when `enable.idempotence` is set to `false`, Kafka does not track gaps in transmitted records and is unable to reorder records or account for retransmissions in the face of errors.

In scenarios where strict order is fundamental to the correctness of the system, and *in the absence of idempotence*, it essential that either `retries` is set to `0` or `max.in.flight.requests.per.connection` is set to `1`. However, the preferred alternative is to set `enable.idempotence` to `true`, which will guard against the reordering problem and also avoid record duplication. This is another example where Kafka's configuration favours performance over correctness.

Enable idempotence

The `enable.idempotence` property, when set to `true`, ensures that —

- Any record queued at the producer will be persisted *at most once* to the corresponding partition;
- Records are persisted in the order specified by the producer; and
- Records are persisted to all in-sync replicas before being acknowledged.

The default value of `enable.idempotence` is `false`.

Enabling idempotence requires `max.in.flight.requests.per.connection` to be less than or equal to `5`, retries to be greater than `0` and `acks` set to `all`. If these values are not explicitly set by the user, suitable values will be chosen by default. If incompatible values are set, a `ConfigException` will be thrown during producer initialisation.

The problem of non-idempotent producers arises when an intermittent error causes a timeout of a record acknowledgement on the return path when `acks` is set to `1` or to `all`, and `retries` is set to a

number greater than zero. In other words, the broker would have received and persisted the record, but the waiting producer times out due to a delay. The producer will resend the record if it has retries remaining, which will result in a second identical copy of the record persisted on the partition at a later offset. As a consequence, all consumers will observe a duplicate record when reading from the topic. Furthermore, due to the batching nature of the producer, it is likely that duplicates will be observed as contiguous record sequences rather than one-off records.

The idempotence mechanism in Kafka works by assigning a monotonically increasing sequence number to each record, which in combination with a unique producer ID (PID), creates a partial ordering relationship that can be easily reconciled at the receiving broker. The broker maintains an internal map of the highest sequence number recorded for each PID, for each partition. A broker can safely discard a record if its sequence number does not exceed the last persisted sequence number by one. If the increment is greater than one, the broker will respond with an `OUT_OF_ORDER_SEQUENCE_NUMBER` error, forcing the batches to be re-queued on the producer. The requirement that changes to the sequence numbers are contiguous proverbially kills two birds with one stone. In addition to ensuring idempotence, this mechanism also guarantees the ordering of records and avoids the reordering issue when `max.in.flight.requests.per.connection` is set to allow multiple outstanding in-flight records.

The deduplication guarantees apply only to the individual records queued within the producer. If the application calls `send()` with a duplicate record, the producer will assume that the records are distinct, and will send the second with a new sequence number. As such, it is the responsibility of the application to avoid queuing unnecessary duplicates.

> The official documentation describes the `enable.idempotence` property as mechanism for the producer to ensure that *exactly one* copy of each record is written in the stream and that records are written in the strict order they were published in.
>
> Without a suitable *a priori* assurance as to the liveness of the producer, the broker, and the reliability of the underlying network, the conjecture in the documentation is inaccurate. The producer is unable to enact any form of assurance if, for example, its process fails. Restarting the process would lose any queued records, as the producer does not buffer these to a stable storage medium prior to returning from `send()`. (The producer's accumulator is volatile.) Also, if the network or the partition leader becomes unavailable, and the outage persists for an extent of time beyond the maximum allowed by the `delivery.timeout.ms` property, the record will time out, yielding a failed result. In this case, the write semantics will be *at most once*.
>
> A degraded network or a slow broker may also present a problem. Suppose a record was published successfully, but the response timed out in such as way as to exhaust the `delivery.timeout.ms` timeout on the producer. The client will return an error to the application, which may either skip the record, or publish it a second time. In the latter case, the producer client will not detect a duplicate, and will publish what is effectively an identical record a second time. In this case, the write semantics will be *at least once*.
>
> Thus, the official documentation should be taken in the context of encountering intermittent errors within an otherwise functioning system, where the system is capable of making progress within all

> of the specified timeouts. If and only if the producer received an acknowledgement of the write from the broker, can we be certain that *exactly-once* write semantics were in force.

In a typical order-preserving application, setting `retries` to `0` is impractical, as it will flood the application with transient errors that could otherwise have been retried. Therefore, capping `max.in.flight.requests.per` to `1` or setting `enable.idempotence` to `true` is the more sensible thing to do, with the latter being the preferred approach, being less impacted by high-latency networks.

Compression type

The `compression.type` controls the algorithm that the producer will use to compress record batches before forwarding them on to the partition leaders. The valid values are:

- `none`: Compression is disabled. This is the default setting.
- `gzip`: Use the *GNU Gzip* algorithm — released in 1992 as a free substitute for the proprietary `compress` program used by early UNIX systems.
- `snappy`: Use Google's *Snappy* compression format — optimised for throughput at the expense of compression ratios.
- `lz4`: Use the *LZ4* algorithm — also optimised for throughput, most notably for the decompression speed.
- `zstd`: Use Facebook's *ZStandard* — a newer algorithm introduced in Kafka 2.1.0, intended to achieve an effective balance between throughput and compression ratios.

This topic is discussed in greater detail in Chapter 12: Batching and Compression[19]. To summarise, compression may offer significant gains in network efficiency. It also reduces the amount of disk I/O and storage space taken up on the brokers.

Key and value serializer

The `key.serializer` and the `value.serializer` properties allow the user to configure the mechanism for serializing the records' keys and values, respectively. These properties have no defaults. An alternative way to specify a serializer is to directly instantiate one and pass it as a reference to an overloaded `KafkaProducer` constructor.

Serialization is a complex field that transcends client configuration, touching on the broader issues of custom data types and application design. This chapter will not discuss the nuances of serialization; instead, consult Chapter 7: Serialization for a comprehensive discussion on this topic.

[19] chapter-batching-compression

Partitioner

The `partitioner.class` property allows the application to override the default partitioning scheme by specifying an implementation of a `org.apache.kafka.clients.producer.Partitioner`. Unless instructed otherwise, the producer will use the `org.apache.kafka.clients.producer.internals.DefaultPartitioner` implementation.

The behaviour of the `DefaultPartitioner` varies depending on the attributes of the record:

1. If a partition is explicitly specified in the `ProducerRecord`, that partition will always be used.
2. If no partition is set, but a key has been specified, the key is hashed to determine the partition number.
3. If neither the partition nor the key is specified, and the current batch already has a 'sticky' partition assigned to it, then maintain the same partition number as the current batch.
4. If neither of the above conditions are met, then assign a new 'sticky' partition to the current batch and use it for the current record.

Points #1 and #2 capture the age-old behaviour of the `DefaultPartitioner`. Points #3 and #4 were added in the 2.4.0 release of Kafka, as part of KIP-480[20]. Previously, the producer would vacuously allocate records to partitions in a round-robin fashion. While this spreads the load evenly among the partitions, it largely negates the benefits of batching. Since partitions are mastered by different brokers in the cluster, this approach used to engage potentially several brokers to publish a batch, resulting in a much higher typical latency, influenced by the slowest broker. The 2.4.0 update limits the engagement to a single broker for any given unkeyed hash, reducing the 99th percentile latency by a factor of two to three, depending on the record throughput. The partitions are still evenly loaded over a long series of batches.

Hashing a key to resolve the partition number is performed by passing the byte contents of the key through a *MurmurHash2* function, then taking the low-order 31 bits from the resulting 32-bit by masking off the highest order bit (bitwise `AND` with `0x7fffffff`). The resulting value is taken, modulo the number of partitions, to arrive at the partition index.

While hashing of record keys and mapping of records to partitions might appear straightforward, it is laden with gotchas. A more thorough analysis of the problem and potential solutions are presented in Chapter 6: Design Considerations. Without going into the details here, the reader is urged to abide by one rule: when the correctness of a system is predicated on the key-centric ordering of records, avoid resizing the topic as this will effectively void any ordering guarantees that the consumer ecosystem may have come to rely upon.

In addition to the `DefaultPartitioner`, the Java producer client also contains a `RoundRobinPartitioner` and a `UniformStickyPartitioner`.

[20] https://cwiki.apache.org/confluence/display/KAFKA/KIP-480%3A+Sticky+Partitioner

The `RoundRobinPartitioner` will forward the record to a user-specified partition if one is set; otherwise, it will indiscriminately distribute the writes to all partitions in a round-robin fashion, regardless of the value of the record's key. Because the allocation of unkeyed records to partitions is nondeterministic, it is entirely possible for records with the same key to occupy different partitions and be processed out of order. This partitioner is useful when an event stream is not subject to ordering constraints, in other words, when Kafka is used as a proverbial 'firehose' of unrelated events. Alternatively, this partitioner may be used when the consumer ecosystem has its own mechanism for reassembling events, which is independent of Kafka's native partitioning scheme.

The `UniformStickyPartitioner` is a pure implementation of KIP-480[21] that was introduced in Kafka 2.4.0. This partitioner will forward the record to a user-specified partition if one is set; otherwise, it will disregard the key, and instead assign 'sticky' partitions numbers based on the current batch.

One other 'gotcha' with partitioners lies in them being a pure producer-side concern. The broker has no awareness of the partitioner used, it defers to the producer to make this decision for each submitted record. This assumes that the producer ecosystem has agreed on a single partitioning scheme and is applying it uniformly. Naturally, if reconfiguring the producers to use an alternate partitioner, one must ensure that this change is rolled out atomically — there cannot be two or more producers concurrently operating with different partitioners.

One can implement their own partitioner, should the need for one arise. Perhaps you are faced with a bespoke requirement to partition records based on the contents of their payload, rather than the key. While a custom partitioner may satisfy this requirement, a more straightforward approach would be to concatenate the order-influencing attributes of the payload into a synthetic key, so that the default partitioner can be used. (It may be necessary to pre-hash the key to cap its size.)

Interceptors

The `interceptor.classes` property enables the application to intercept and potentially mutate records en route to the Kafka cluster, just prior to serialization and partition assignment. This list is empty by default. The application can specify one or more interceptors as a comma-separated list of `org.apache.kafka.clients.producer.ProducerInterceptor` implementation classes. The `ProducerInterceptor` interface is shown below, with the Javadoc comments removed for brevity.

[21] https://cwiki.apache.org/confluence/display/KAFKA/KIP-480%3A+Sticky+Partitioner

```
public interface ProducerInterceptor<K, V> extends Configurable {
  public ProducerRecord<K, V> onSend(ProducerRecord<K, V> record);

  public void onAcknowledgement(RecordMetadata metadata,
                                Exception exception);

  public void close();
}
```

The `Configurable` super-interface enables classes instantiated by reflection to take configuration parameters:

```
public interface Configurable {
  void configure(Map<String, ?> configs);
}
```

Interceptors act as a plugin mechanism, enabling the application to inject itself into the publishing (and acknowledgement) process without directly modifying the application code.

This naturally leads to a question: Why would anyone augment the publisher using obscurely-configured interceptors, rather than modifying the application code to address these additional behaviours directly in the application code?

Interceptors add an Aspect-Oriented Programming (AOP) style to modelling producer behaviour, allowing one to uniformly address cross-cutting concerns at independent producers in a manner that is modular and reusable. Some examples that demonstrate the viability of AOP-style interceptors include:

- Accumulation of producer metrics — tracking the total number of records published, records by category, etc.
- End-to-end tracing of information flow through the system — using correlation headers present in records to establish a graph illustrating the traversal of messages through the relaying applications, identifying each intermediate junction and the timings at each point.
- Logging of entire record payloads or a subset of the fields in a record.
- Ensuring that outgoing records comply with some schema contract.
- Data leak prevention — looking for potentially sensitive information in records, such as credit card numbers or JWT bearer tokens.

Once defined and tested in isolation, this behaviour could then be encompassed in a shared library and applied to any number of producers.

There are several caveats to implementing an interceptor:

- Runtime exceptions thrown from the interceptor will be caught and logged, but will not be allowed to propagate to the application code. As such, it is important to monitor the client logs when writing interceptors. A trapped exception thrown from one interceptor has no bearing on the next interceptor in the list: the latter will be invoked after the exception is logged.
- An interceptor may be invoked from multiple threads, potentially concurrently. The implementation must therefore be thread-safe.
- When multiple interceptors are registered, their `onSend()` method will be invoked in the order they were specified in the `interceptor.classes` property. This means that interceptors can act as a transformation pipeline, where changes to the published record from one interceptor can be fed as an input to the next. This style of chaining is generally discouraged, as it leads to *content coupling* between successive interceptor implementations — generally regarded as the worst form of coupling and leading to brittle code. An exception in one interceptor will not abort the chain — the next interceptor will still be invoked without having the previous transformation step applied, breaking any assumptions it may have as to the effects of the previous interceptor.
- The `onAcknowledgement()` method will be invoked by the I/O thread of the producer. This blocks the I/O thread until the method returns, preventing it from processing other acknowledgements. As a rule of thumb, the implementation of `onAcknowledgement()` should be reasonably fast, returning without unnecessary delays or blocking. It should ideally avoid any time-consuming I/O of its own. Any time-consuming operations on acknowledged records should be delegated to background threads.

In light of the above, `ProducerInterceptor` implementations should be simple, fast, standalone units of code that maintain minimal state, with no dependencies on one another. Any non-trivial interceptor implementation should have a mandatory exception handler surrounding the bodies of `onSend()` and `onAcknowledgement()`, so as to control precisely what happens in the event of an error.

Maximum block time

The `max.block.ms` configuration property controls how long `KafkaProducer.send()` and `KafkaProducer.partitions` will block for. These methods can be blocked for two reasons: either the internal accumulator buffer is full or the metadata required for their operation is unavailable. The default value is `60000` (one minute). Blocking in the user-supplied serializers or partitioner will not be counted against this timeout.

Batch size and linger time

The `batch.size` and `linger.ms` properties collectively control the extent to which the producer will attempt to batch queued records in order to maximise the outgoing transmission efficiency. The default values of `batch.size` and `linger.ms` are `16384` (16 KiB) and `0` (milliseconds), respectively.

The `linger.ms` setting induces batching in the absence of heavy producer traffic by adding a small amount of artificial delay — rather than immediately sending a record the moment it is enqueued,

the producer will wait for up to a set delay to allow other records to accumulate in a batch. This maximises the amount of data that can be transmitted in one go. Although records may be allowed to linger for up to the duration specified by `linger.ms`, the `batch.size` property will have an overriding effect, dispatching the batch once it reaches the set maximum size. Another way of looking at it: while the `linger.ms` property only comes into the picture when the producer is lightly loaded, the `batch.size` property is continuously in effect, ensuring the batch never grows above a set cap.

This topic is discussed in greater detail in Chapter 12: Batching and Compression. To summarise, batching improves network efficiency and throughput, at the expense of increasing publishing latency. It is often used collectively with compression, as the latter is more effective in the presence of batching.

Request timeout

The `request.timeout.ms` property controls the maximum amount of time the client will wait for a broker to respond to an in-flight request. If the response is not received before the timeout elapses, the client will either resend the request if it has a retry budget available (configured by the `retries` property), or otherwise fail the request. The default value of `request.timeout.ms` is `30000` (30 seconds).

Delivery timeout

The `delivery.timeout.ms` property sets an upper bound on the time to report success or failure after a call to `send()` returns, having a default value of `120000` (two minutes).

This setting acts as an overarching limit, encompassing —

- The time that a record may be delayed prior to sending;
- The time to await acknowledgement from the broker (if `acks=1` or `acks=all`); and
- The time budgeted for retryable send failures.

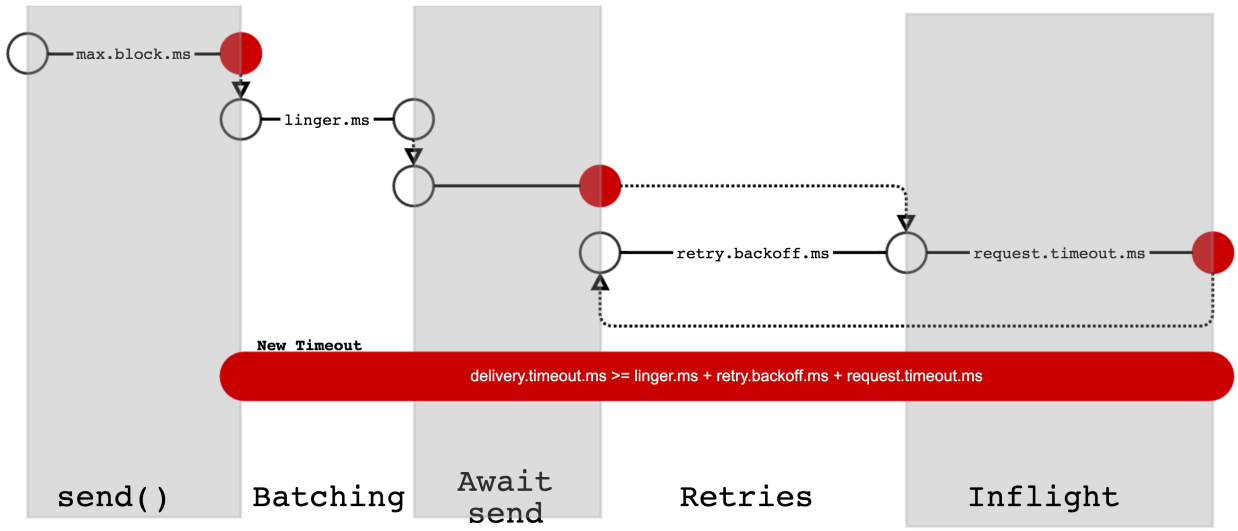
The producer may report failure to send a record earlier than this time if either an unrecoverable error is encountered, the retries have been exhausted, or the record is added to a batch which reached an earlier delivery expiration deadline. The value of this property should be greater than or equal to the sum of `request.timeout.ms` and `linger.ms`.

This property is a relatively recent addition, introduced in Kafka 2.1.0 as part of KIP-91[22]. The main motivation was to consolidate the behaviour of several related configuration properties that could potentially affect the time a record may be in a pending state, and thereby inadvertently extend this time beyond the application's tolerance to obtain a successful or failed outcome of publishing the record. The consolidated `delivery.timeout.ms` property acts as an overarching budget on the total pending time, terminating the publishing process and yielding an outcome at or just before this

[22] https://cwiki.apache.org/confluence/display/KAFKA/KIP-91+Provide+Intuitive+User+Timeouts+in+The+Producer

time is expended. In doing so, it does not deprecate the underlying properties. In fact, it prolongs their utility by making them safer to use in isolation, allowing for a more confident fine-tuning of producer behaviour.

The diagram below, lifted from the description of KIP-91[23], illustrates the `delivery.timeout.ms` property and its relation to the other properties that it subsumes. The red circles indicate points where a timeout may occur.



Delivery timeout behaviour

The individual stages that constitute the record's journey from the producer to the broker are explained below.

- The initial call to `send()` can block up to `max.block.ms`, waiting on metadata or queuing for available space in the producer's accumulator. Upon completion, the record is appended to a batch.
- The batch becomes eligible for transmission over the wire when either `linger.ms` or `batch.size` has been reached.
- Once the batch is ready, it must wait for a transmission opportunity. A batch may be sent when all of the following conditions are met:
  - The metadata for the partition is known and the partition leader has been identified;
  - A connection to the leader exists; and
  - The current number of in-flight requests is less than the number specified by `max.in.flight.requests.per`
- Once the batch is transmitted, the `request.timeout.ms` property limits the time that the producer will wait for an acknowledgement from the partition leader.
- If the request fails and the producer has one or more retries remaining, it will attempt to send the batch again. Each send attempt will reset the request timeout, in other words, each retry gets its own `request.timeout.ms`.

[23] https://cwiki.apache.org/confluence/display/KAFKA/KIP-91+Provide+Intuitive+User+Timeouts+in+The+Producer

The *await-send* stage is the most troublesome section of the record's journey, as there is no way to precisely determine how long a record will spend in this state. Firstly, the batch could be held back by an issue in the cluster, beyond the producer's control. Secondly, it may be held back by the preceding batch, which is particularly likely when the `max.in.flight.requests.per.connection` property is set to `1`.

Prior to the Kafka 2.1.0, time spent in the *await-send* stage used to be bounded by the same transmission timeout that is used after the batch is sent — `request.timeout.ms`. The producer would eagerly start the transmission clock when the record entered *await-send*, even though technically it was still queued on the producer. Records blocked waiting for a metadata refresh or for a prior batch to complete could be pessimistically expired, even though it was possible to make progress. This problem was compounded if the prior batch was stuck in the sending stage, which granted it additional time credit — amplified by the value of the `retries` property. In other words, it was possible for an *await-send* batch to time out *before* its immediate forerunner, if the preceding batch happened to have made more progress.

The strength of the `delivery.timeout.ms` property is that it does not discriminate between the various stages of a record's journey, nor does it unfairly penalise records due to contingencies in a preceding batch.

Transactional ID and transaction timeout

The `transactional.id` and `transaction.timeout.ms` properties alter the behaviour of the producer with respect to transactions. Transactions would be classed as a relatively advanced topic on the Kafka 'complexity' spectrum; please consult Chapter 18: Transactions for a more in-depth discussion.

Consumer configuration

This section describes configuration options that are specific to the consumer client type. Some configuration properties are reciprocals of their producer counterparts; these will be covered first.

Key and value deserializer

Analogously to the producer configuration, the `key.deserializer`, and the `value.deserializer` properties specify the mechanism for deserializing the records' keys and values, respectively. As per the producer scenario, a user can alternatively instantiate the deserializers directly and pass them as references to an overloaded `KafkaConsumer` constructor.

A broader discussion of (de)serialization is presented in Chapter 7: Serialization. The material presented in that chapter should be consulted prior to implementing custom (de)serializers.

Interceptors

The `interceptor.classes` property is analogous to its producer counterpart, specifying a comma-separated list of `org.apache.kafka.clients.consumer.ConsumerInterceptor` implementations, allowing for the inspection and possibly mutation of records before a call to `Consumer.poll()` returns them to the application.

The `ConsumerInterceptor` interface is shown below, with the Javadoc comments removed for brevity.

```
public interface ConsumerInterceptor<K, V>
    extends Configurable, AutoCloseable {
  public ConsumerRecords<K, V>
      onConsume(ConsumerRecords<K, V> records);

  public void onCommit(Map<TopicPartition,
                       OffsetAndMetadata> offsets);

  public void close();
}
```

The use of interceptors on the consumer follows the same rationale as we have seen on the producer. Specifically, interceptors act as a plugin mechanism, offering a way to uniformly address cross-cutting concerns at independent consumers in a manner that is modular and reusable.

There are similarities and differences between the producer and consumer-level interceptors. Beginning with the similarities:

- Runtime exceptions thrown from the interceptor will be caught and logged, but will not be allowed to propagate to the application code. A trapped exception thrown from one interceptor has no bearing on the next interceptor in the list: the latter will be invoked after the exception is logged.
- When multiple interceptors are registered, their `onConsume()` method will be invoked in the order they were specified in the `interceptor.classes` property, allowing interceptors to act as a transformation pipeline. The same caveat applies as per the producer scenario; this style of chaining is discouraged, as it leads to *content coupling* between successive interceptor implementations — resulting in brittle code.
- The `onCommit()` method may be invoked from a background thread; therefore, the implementation should avoid unnecessary blocking.

As for the differences, there is one: unlike `KafkaProducer`, the `KafkaConsumer` implementation is not thread-safe — the shared use of `poll()` from multiple threads is forbidden. Therefore, there is no requirement that a `ConsumerInterceptor` implementation must be thread-safe.

Controlling the fetch size

When retrieving records from a Kafka cluster, one ultimately needs to decide how much data is enough, and how much is too much. More is not always better; while increasing the fetch size will lead to improved network utilisation and therefore higher throughput, it comes at a price — the end-to-end propagation delay will suffer as a result. Conversely, fetching just a few records will make the application more responsive, but small fetches require more round-trips to move the same amount of data, negatively impacting the throughput.

Tuning the fetch size is among the performance-impacting decisions one has to face when building consumer applications. Kafka does little to help simplify this process. There is no consolidated 'throughput ⊠ latency' dial that one can adjust to their satisfaction; instead, there are numerous consumer settings that collectively impact the fetch behaviour.

The most apparent control is the `timeout` parameter to the `Consumer.poll()` method. However, this is only an upper bound on the time that the method call will block for; its effects are limited to the consumer — it does not limit the amount of data retrieved from the brokers. There are several other consumer properties that influence the fetching of data; their influence extends past the client behaviour, affecting how the brokers respond to fetch queries.

The first pair of properties is the `fetch.min.bytes` and `fetch.max.bytes`. Respectively, these properties constrain the minimum and the maximum amount of data the broker should return for a fetch request.

If insufficient data is available, the request will wait for the quantity of data specified by `fetch.min.bytes` to accumulate before answering the request. The default setting is `1`, meaning that a broker will respond as soon as a single byte is available, or if the fetch request times out. The latter is governed by a separate but complementary `fetch.max.wait.ms` property, which defaults to `500` (milliseconds).

The `fetch.max.bytes` property sets a *soft* upper bound on the fetch request, acting more as a guide than a limit. Records are written and fetched in batches, which are treated as indivisible units from a broker's perspective. Considering that the size of a single record might conceivably eclipse the `fetch.max.bytes` limit, and indeed, the size of a given batch might also be correspondingly larger, the fetch mechanism allows for this — potentially returning a larger batch than what was specified by `fetch.max.bytes`. In doing so, it allows the consumer to make progress, which would otherwise be indefinitely obstructed had the `fetch.max.bytes` limit been enforced verbatim. More accurately, the query permits an 'oversized' batch if it is the first batch in the first non-empty partition. The default value of `fetch.max.bytes` is `52428800` (50 MiB).

The reason that batches are not broken up into individual records and returned in sub-batch quantities is due to Kafka's fundamental architecture and deliberate design decisions that contribute to its performance characteristics. Batches are an end-to-end construct; formed by the producer, they are transported, persisted, and delivered to the consumers as-is — without unpacking the batch or inspecting its contents — minimising unnecessary work on the broker. In the best-case scenario, a batch is persisted and retrieved using *zero-copy*, where transfer operations between the network and the storage devices occur with no involvement from the CPU. Brokers are often the point of contention — they form a static topology that does not scale elastically — unlike, say, consumers. Reducing their workload by imparting more work onto the producer and consumer clients leads to a more scalable system.

A further refinement of `fetch.max.bytes` is the `max.partition.fetch.bytes` property, applying a soft limit on a per-partition basis. The default value of `max.partition.fetch.bytes` is `1048576` (1 MiB).

There are no official guidelines for tuning Kafka with respect to `fetch.max.bytes` and `max.partition.fetch.bytes`. While their individual functions are clear, their mutual relationship is not as apparent. One must consider what happens when a fetch response aggregates batches from multiple partitions. Suppose a topic is unevenly loaded, where relatively few partitions collectively carry more records than the remaining majority of partitions. If the `max.partition.fetch.bytes` setting is overly relaxed, the results of the fetch will be biased towards the heavily-loaded partitions. In other words, the fetch quota set by `fetch.max.bytes` will be disproportionately exhausted by the minority partitions. In the best case, this will negatively affect the propagation latencies of the majority partitions; in the worst case, this might lead to periods of starvation, where records for certain partitions are unceasingly dropped from the response.

This *Pareto Effect* is actually more common than one might imagine. As record keys *tend* to reflect the identifiers of real-world entities, the distribution of records within a Kafka topic often acquires an uncanny resemblance to the real world, which as we know, is often accurately described by the *power law*. Setting a conservative value for `max.partition.fetch.bytes` improves fairness, increasing the likelihood of aggregating data over the majority partitions by penalising heavily loaded partitions. However, an overly conservative value undermines the allowance set by `fetch.max.bytes`. Furthermore, it may lead to a buildup of records in minority topics.

Left to its devices, this discussion leads to broader topics, such as queuing and quality of service, and further still, to subjects such as economics, politics, and philosophy, which are firmly outside the scope of this text. The relative tuning of the fetch controls can be likened to the redistribution of wealth. It can only be said that the decision to favour one group over another (in the context of Kafka's topic partitioning, of course) must stem from the non-functional requirements of the application, rather than some hard and fast rule.

The final configuration property pertinent to this discussion is `max.poll.records`, which sets the upper bound on the number of records returned in a single call to `poll()`. Unlike some of the other properties that control the fetch operation on the broker, and akin to the poll timeout, the effects

of this property are confined to the client. After receiving the record batches from the brokers, and having unpacked the batches, the consumer will artificially limit the number of records returned. The excluded records will remain in the fetch buffer — to be returned in a subsequent call to `Consumer.poll()`. The default value of `max.poll.records` is `500`.

> The original motivation for artificially limiting the number of returned records was largely historical, revolving around the behaviour of the `session.timeout.ms` property at the time. The `max.poll.records` property was introduced in version 0.10.0.0 of Kafka, described in detail in KIP-41[a].
>
> The act of polling a Kafka cluster did not just retrieve records — it had the added effect of signalling to the coordinator that the consumer is in a healthy state and able to handle its share of the event stream. Given a combination of a sufficiently large batch and a high time-cost of processing individual records, a consumer's poll loop might have taken longer than the deadline enforced by the `session.timeout.ms` property. When this happened, the coordinator would assume that the consumer had 'given up the ghost', so to speak, and reassign its partitions among the remaining consumers in the encompassing consumer group. In reducing the number of records returned from `poll()`, the application would effectively slacken its processing obligations between successive polls, increasing the likelihood that a cycle would complete before the `session.timeout.ms` deadline elapsed.
>
> A second change was introduced in version 0.10.1.0 and is in effect to this day; the behaviour of polling with respect to consumer liveness was radically altered as part of KIP-62[b]. The changes saw consumer heartbeating extracted from the `poll()` method into a separate background thread, invoked automatically at an interval not exceeding that of `heartbeat.interval.ms`. Regular polling is still a requisite for proving that a consumer is healthy; however, the poll deadline is now *locally* enforced on the consumer using the `max.poll.interval.ms` property as the upper bound. If the application fails to poll within this period, the client will simply stop sending heartbeats. This change fixed the root cause of the problem — conflating the cycle time with heartbeating, resulting in either compromising on failure detection time or faulting a consumer prematurely, inadvertently creating a scenario where two consumers might simultaneously handle the same records.
>
> The change to heartbeating markedly improved the situation with respect to timeouts, but it did not address all issues. There is no reliable way for the outgoing consumer to determine that its partitions were revoked — not until the next call to `poll()`. By then, any uncommitted records may have been replayed by the new consumer — resulting in the simultaneous processing of identical records by two consumers — each believing that they 'own' the partitions in question — a highly undesirable scenario in most stream processing applications. The use of a `ConsumerRebalanceListener` does not help in this scenario, as the rebalance callbacks are only invoked from within a call to `poll()`, using the application's polling thread – the same thread that is overwhelmed by the record batch.
>
> To be clear, the improvements introduced in Kafka 0.10.1.0 have not eliminated the need for `max.poll.records`. Particularly when the average time-cost of processing a record is high, limiting the number of in-flight records is still essential to a predictable, time-bounded poll loop. In the absence of this limit, the number of returned records could still backlog the consumer, breaching the deadline set by `max.poll.interval.ms`. The combination of the `max.poll.records` and the more recent `max.poll.interval.ms` settings should be used to properly manage consumer liveness.

For a deeper understanding of how Kafka addresses the liveness and safety properties of the consumer ecosystem, consult Chapter 15: Group Membership and Partition Assignment.

[a] https://cwiki.apache.org/confluence/display/KAFKA/KIP-41%3A+KafkaConsumer+Max+Records
[b] https://cwiki.apache.org/confluence/display/KAFKA/KIP-62%3A+Allow+consumer+to+send+heartbeats+from+a+background+thread

Group ID

The `group.id` property uniquely identifies the encompassing consumer group, and is integral to the topic subscription mechanism used to distribute partitions among consumers. The `KafkaConsumer` client will use the configured group ID when its `subscribe()` method is invoked. The ID can be up to 255 characters in length, and can include the following characters: `a-z`, `A-Z`, `0-9`, `.` (period), `_` (underscore), and `-` (hyphen). Consumers operating under a consumer group are fully governed by Kafka; aspects such as basic availability, load-balancing, partition exclusivity, and offset persistence are taken care of.

This property does not have a default value. If unset, a *free consumer* is presumed. Free consumers do not subscribe to a topic; instead, the consuming application is responsible for manually assigning a set of topic-partitions to the consumer, individually specifying the starting offset for each topic-partition pair. *Free consumers do not commit their offsets to Kafka*; it is up to the application to track the progress of such consumers and persist their state as appropriate, using a data store of their choosing. The concepts of automatic partition assignment, rebalancing, offset persistence, partition exclusivity, consumer heartbeating and failure detection (liveness, in other words), and other so-called 'niceties' accorded to consumer groups cease to exist in this mode.

The use of the nominal expression *'free consumer'* to denote a consumer without an encompassing group is a coined term. It is not part of the standard Kafka nomenclature; indeed, there is no widespread terminology that marks this form of consumer.

Group instance ID

The `group.instance.id` property specifies a long-term, stable identity for the consumer instance — allowing it to act as a static member of a group. This property is optional; if set, the group instance ID is a non-empty, free-form string that must be unique within the consumer group.

Static group membership is described in detail in Chapter 15: Group Membership and Partition Assignment. The reader is urged to consult this chapter if contemplating the use of static group membership, or mixing static and dynamic membership in the same consumer group.

As an outline, static membership is used in combination with a larger `session.timeout.ms` value to avoid group rebalances caused by transient unavailabilities, such as intermediate failures and process

restarts. Static members join a group much like their dynamic counterparts and receive a share of partitions. However, when a static member leaves, the group leader preserves the member's partition assignments, irrespective of whether the departure was planned or unintended. The affected partitions are simply parked; there is no reassignment, and, consequently, the partitions will begin to accumulate lag. Upon its eventual return, the bounced member will resume the processing of its partitions from its last committed point. Static membership aims to lessen the impact of rebalancing at the expense of individual partition availability.

Heartbeat interval, session timeout, and the maximum poll interval

The `heartbeat.interval.ms`, `session.timeout.ms`, and `max.poll.interval.ms` properties are closely intertwined, collectively controlling Kafka's failure detection behaviour. This behaviour only applies to consumers operating within a group; free consumers are not subject to health checks.

The topic of failure detection and liveness of the consumer ecosystem is covered in Chapter 15: Group Membership and Partition Assignment. The reader will be advised that this topic ranks high on the 'gotcha' spectrum; so much so that the incorrect use of Kafka's failure detection capabilities will jeopardise the correctness of the system, leading to stalled consumers or state corruption.

The following is a highly condensed summary of these properties and their effects.

The `heartbeat.interval.ms` property controls the frequency with which the `KafkaConsumer` client will automatically send heartbeats to the coordinator, indicating that its process is alive and can reach the cluster. On its end, the group coordinator will allow for up to the value of `session.timeout.ms` to receive the heartbeat; failure to receive a heartbeat within the set deadline will result in the forceful expulsion of the consumer from the group, and the reassignment of the consumer's partitions. (This is true for both static and dynamic consumers.)

The `max.poll.interval.ms` stipulates the maximum delay between successive invocations of `poll()`, enforced internally by the `KafkaConsumer`. For dynamic consumers, if the poll-process loop fails to poll in time, the consumer client will cease to send heartbeats and will proactively leave the group — promptly causing a rebalance on the coordinator. For static consumers, a missed deadline will result in the quiescing of heartbeats, but no leave request is sent; it will be up to the coordinator to evict a failed consumer if the latter fails to reappear within the `session.timeout.ms` deadline.

The table below lists the default values of these properties.

Property	Default value
`heartbeat.interval.ms`	`3000` (3 seconds)
`session.timeout.ms`	`10000` (10 seconds)
`max.poll.interval.ms`	`300000` (5 minutes)

While the effects of these three properties are abundantly documented and clear, the idiosyncrasies of the associated failure recovery apparatus and the implications of the numerous edge cases remain a mystery to most Kafka practitioners. To avoid getting caught out, the reader is urged to study

Chapter 15: Group Membership and Partition Assignment.

Auto offset reset

The `auto.offset.reset` property stipulates the behaviour of the consumer when no prior committed offsets exist for the partitions that have been assigned to it, or if the specified offsets are invalid.

From the perspective of a consumer acting within an encompassing group, the absence of valid offsets may be observed in three scenarios:

1. When the group is initially formed and the lack of offsets is to be expected. This is the most intuitive and distinguishable scenario and is largely self-explanatory.
2. When an offset for a particular partition has not been committed for a period of time that exceeds the configured retention period of the `__consumer_offsets` topic, and where the most recent offset record has subsequently been truncated.
3. When a committed offset for a partition exists, but the location it points to is no longer valid.

To elaborate on the second scenario: in order to commit offsets, a consumer will send a message to the group coordinator, which will cache the offsets locally and also publish the offsets to an internal topic named `__consumer_offsets`. Other than being an internal topic, there is nothing special about `__consumer_offsets` — it behaves like any other topic in Kafka, meaning that it will eventually start shedding old records. The offsets topic has its retention set to seven days by default. (This is configurable via the `offsets.retention.minutes` broker property.) After this time elapses, the records become eligible for collection.

Prior to Kafka 2.0.0, the default retention period of `__consumer_offsets` was 24 hours, which confusingly did not align with the default retention period of seven days for all other topics. This used to routinely catch out unsuspecting users; keeping a consumer offline for a day was all it took to lose your offsets. One could wrap up their work on a Friday, come back on the following Monday and *occasionally* discover that their offsets had been reset. The confusion was exacerbated by the way topic retention works — lapsed records are not immediately purged, they only become candidates for truncation. The actual truncation happens when a log segment file is closed, which cannot be easily predicted as it depends on the amount of data that is written to the topic. KIP-186[24] addressed this issue for release 2.0.0.

The third scenario occurs as a result of routine record truncation, combined with a condition where at least one persisted offset refers to the truncated range. This may happen when the topic in question has shorter retention than the `__consumer_offsets` topic — as such, the committed offsets outlive the data residing at those offsets.

The offset reset consideration is not limited to consumer groups. A *free* consumer — one that is operating without an encompassing consumer group — can experience an invalid offset during a

[24] https://cwiki.apache.org/confluence/display/KAFKA/KIP-186%3A+Increase+offsets+retention+default+to+7+days

call to `Consumer.seek()`, if the supplied offset lies outside of the range bounded by the low-water and high-water marks.

Whatever the reason for the missing or invalid offsets, the consumer needs to adequately deal with the situation. The `auto.offset.reset` property lets the consumer select from one of three options:

- `earliest`: Reset the consumer to the low-water mark of the topic — the offset of the first retained record for each of the partitions assigned to the consumer where no committed offset exists.
- `latest`: Reset the consumer to the high-water mark — the offset immediately following that of the most recently published record for each relevant partition. This is the default option.
- `none`: Do not attempt to reset the offsets if any are missing; instead, throw a `NoOffsetForPartitionException`.

Resetting the offset to `latest`, being the default setting, has the potential to cause havoc, as it runs contrary to Kafka's at-least-once processing tenet. If a consumer group were to lose committed offsets following a period of downtime, the resulting reset would see the consumers' read positions 'jump' instantaneously to the high-water mark, skipping over all records following the last committed point (inclusive of it). Any lag accumulated by the consumer would suddenly disappear. The delivery characteristics for the skipped records would be reduced to 'at most once'. Where consumers request a subscription under a consumer group, it is highly recommended that the default offset scheme is set to `earliest`, thereby maintaining at-least-once semantics for as long as the consumer's true lag does not exceed the retention of the subscribed topic(s).

Enable auto-commit and the auto-commit interval

There `enable.auto.commit` property controls whether automatic offset committing should be enabled for grouped consumers. The default setting is `true`, which activates the periodic background committing of offsets. This process starts from the point when the application subscribes to one or more topics by invoking one of the overloaded `subscribe()` methods. When enabled, the `auto.commit.interval.ms` property controls the interval of the background auto-commit task, which is set to `5000` (5 seconds) by default. The auto-commit scope encompasses the offsets of the records returned during the most recent call to `poll()`.

The above narrative reflects the official Kafka documentation, but there is more to it. Taking the above for gospel, the more cautious among us might spot a problem. Namely, if auto-commit unconditionally commits offsets every five seconds (or whatever the interval has been set to), what happens to the in-flight records that are yet to be processed? Would they be inadvertently committed, and wouldn't that violate the at-least-once processing semantics?

Indeed, practitioners are mostly divided into two camps: the majority, who have remained oblivious to this concern, and the remaining minority, who have expressed their discomfort with Kafka's default approach. The real answer is somewhat paradoxical. Although it would appear that there is

a critical flaw in the consumer's design, and, indeed, the documentation seems to support this theory, the implementation compensates for this in a subtle and surreptitious manner. Whilst the documentation states that a commit will occur in the background at an interval specified by the configuration, the implementation relies on the application's poll-process to initiate the commit from within the `poll()` method, rather than tying an auto-commit action directly to the system time. Furthermore, the auto-commit only occurs if the last auto-commit was longer than `auto.commit.interval.ms` milliseconds ago. By committing from the processing thread, and *provided record processing is performed synchronously from the poll-process loop*, the `KafkaConsumer` implementation *will not* commit the offsets of in-flights records, standing by its at-least-once processing vows.

While the above explanation might appear reassuring at first, consider the following: the present behaviour is implementation-specific and unwarranted. It is not stated in the documentation, nor in the Javadocs, nor in the KIPs. As such, there is no commitment, implied or otherwise, on Kafka's maintainers to honour this behaviour. Even a minor release could, in theory, move the auto-commit action from a poll-initiated to a timer-driven model. If the reader is concerned at the prospect of this occurring, it may be prudent to disable the offset auto-commit feature and to always commit the offsets manually using `Consumer.commitAsync()`.

Another implication of the offset auto-commit feature is that it extends the window of uncommitted offsets beyond the set of in-flight records. Whilst this is also true of `Consumer.commitAsync()` to a degree, auto-commit will further compound the delay — up to the value of `auto.commit.interval.ms`. With that in mind, asynchronous manual committing is preferred if the objective is to reduce the persisted offset lag while maintaining a decent performance profile. If, on the other hand, the objective is to curtail the persisted offset lag at any cost, the use of the synchronous `Consumer.commitSync()` method would be most appropriate. The latter may be fitting when the average time-cost of processing a record is high, and so the replaying of records is highly undesirable.

Enabling offset auto-commit may have a minor performance benefit in some cases. If records are 'cheap' to process (in other words, record handling is not resource-intensive), the poll-process cycle will be short, and manual committing will occur frequently. This results in frequent commit messages and increased bandwidth utilisation. By setting a minimum interval between commits, the bandwidth efficiency is improved at the expense of a longer uncommitted window. Of course, a similar effect may be achieved with a simple conditional expression that checks the last commit time and only commits if the offsets are stale. Having committed the offsets, it updates the last commit time (a simple local variable) for the next go-round.

Partition assignment strategy

The `partition.assignment.strategy` property specifies a comma-separated list of `org.apache.kafka.clients.cons` implementations (in the order of preference) that should be used to orchestrate partition assignment among members of a consumer group. The default value of this property is `org.apache.kafka.clients.consumer.Ra` which assigns contiguous partition ranges to the members of the group.

A comprehensive discussion of this topic is presented in Chapter 15: Group Membership and Partition Assignment. In summary, partition assignment occurs on one of the members of the group — the group leader. In order for assignment to proceed, members must agree on a common assignment strategy — constrained by the assignors in the intersection of the (ordered) sets of assignors across all members.

The reader would have picked up on such terms as 'group leader' and 'group coordinator' throughout the course of this chapter. These refer to different entities. The group leader is a consumer client that is responsible for performing partition assignment. On the other hand, the group coordinator is a broker that arbitrates group membership.

Changing assignors can be tricky, as the group must always agree on at least one assignor. When migrating from one assignor to another, start by specifying both assignors (in either order) in `partition.assignment.strategy` and bouncing consumers until all members have joined the group with both assignors. Then perform the second round of bouncing, removing the outgoing assignor from `partition.assignment.strategy`, leaving only the preferred assignor upon the conclusion of the round. When migrating from the default 'range' assignor, make sure it is added explicitly to the `partition.assignment.strategy` list prior to performing the first round of bounces.

Transactions

The `isolation.level` property controls the visibility of records written within a *pending* transaction scope — whereby a transaction has commenced but not yet completed. The default isolation level is `read_uncommitted`, which has the effect of returning *all* records from `Consumer.poll()`, irrespective of whether they form part of a transaction, and if so, whether the transaction has been committed.

Conversely, the `read_committed` isolation mode will return all non-transactional records, as well as those transactional records where the encompassing transaction has successfully been committed. Because the `read_committed` isolation level conceals any pending records from `poll()`, it will also conceal all following records, irrespective of whether they are part of a transaction — to maintain strict record order from a consumer's perspective.

For a more in-depth discussion on Kafka transactions, the reader may consult Chapter 18: Transactions.

Admin client configuration

The admin client does not have any unique configuration properties of its own; the properties it employs are shared with the producer and consumer clients.

There is a difference in the construction of a `KafkaAdminClient`, compared to its `KafkaProducer` and `KafkaConsumer` siblings. The latter are instantiated directly using a constructor, as we have seen in the examples thus far. `KafkaAdminClient` does not expose a public constructor. Instead, the

`AdminClient` abstract base class offers a static factory method for instantiating a `KafkaAdminClient`. The `AdminClient` is a more recent addition to the Kafka client family, appearing in version `0.11.0.0`, and seems to have taken a different stylistic route compared to its older siblings. (At the time of writing, a static factory method yet to be retrofitted to the `Producer` and `Consumer` interfaces.)

The `AdminClient` interface is rapidly evolving; every significant Kafka release typically adds new capabilities to the admin API. The `AdminClient` interface is marked with the `@InterfaceStability.Evolving` annotation. Its presence means that the API is not guaranteed to maintain backward compatibility across a minor release. From the Javadocs:

```
/**
 * The administrative client for Kafka, which supports managing
 *  and inspecting topics, brokers, configurations and ACLs.
 *
 * ... omitted for brevity ...
 *
 * This client was introduced in 0.11.0.0 and the API is still
 * evolving. We will try to evolve the API in a compatible
 * manner, but we reserve the right to make breaking changes in
 * minor releases, if necessary. We will update the
 * {@code InterfaceStability} annotation and this notice once the
 * API is considered stable.
 */
```

This chapter has taken the reader on a scenic tour of client configuration. There is a lot of it, and almost every setting can materially impact the client. While the number of different settings might appear overwhelming, there is a method to this madness: Kafka caters to varying event processing scenarios, each requiring different client behaviour and potentially satisfying contrasting non-functional demands.

Thorough knowledge of the configuration settings and their implications is essential for both the *effective* and *safe* use of Kafka. This is where the official documentation fails its audience in many ways — while the individual properties are documented, the implications of their use and their various behavioural idiosyncrasies are often omitted, leaving the user to fend for themselves. The intent of this chapter was to demystify these properties, giving the reader immense leverage from prior research and analysis; ideally, learning from the mistakes of others, as opposed to their own.

Chapter 11: Robust Configuration

Chapter 10: Client Configuration covered all aspects of client configuration in detail. Among other points, it was mentioned that when a client is instantiated, it verifies that the supplied configuration is valid. In other words, it checks that the given keys correspond to valid configuration property names that are supported in the context of that client type. Failing to meet this requirement will result in a warning message being emitted via the configured logger, but the client will still launch.

Kafka's cavalier approach to configuration raises the following questions: How does one make sure that the supplied configuration is valid *before* launching the client application? Or must we wait for the application to be deployed before being told that we mucked something up?

Using constants

The most common source of misconfiguration is a simple typo. Depending on the nature of the misspelled configuration entry, there may be a substantial price to pay for getting it wrong. The example presented in Chapter 10: Client Configuration, involving `max.in.flight.requests.per.connection`, suggests that a mistake may incur the loss of record order under certain circumstances. Relying solely on inspecting log files does not inspire a great deal of confidence — the stakes are too high. There must be a way of nailing the property names; knowing up-front what you give the client will actually be used; and knowing early, before any harm is done.

Kafka does not yet have a satisfactory answer to this, nor is there a KIP in the pipeline that aims to solve this. As a concession, Kafka can meet you halfway with constants:

```
Map<String, Object> config =
    Map.of(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
           "localhost:9092",
           ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
           StringSerializer.class.getName(),
           ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
           StringSerializer.class.getName(),
           ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
           1);
```

In the listing above, property names were replaced with static constants. These constants are stable and will not be changed between release versions. So you can be certain that property names have not been misspelled *if they were embedded in code*. The last point is crucial, as properties may be loaded from an external source, as it is often the case. Constants will not guard against this.

Constants will also fail to protect the user in those scenarios where one might accidentally add a constant from `ConsumerConfig` into a producer configuration, or *vice versa*. Admittedly, this is less likely than misspelling a key, but it is still something to be wary of.

Type-safe configuration

When bootstrapping Kafka client configuration from code or loading configuration artifacts from an external source, the recommended approach is to craft dedicated Java classes with strongly typed values, representing the complete set of configuration items that one can reasonably expect to be supplied when their application is run. The external configuration documents can then be mapped to an object form using a JSON or YAML parser. It is also possible to use a plain `.properties` file if the configuration structure is flat. Alternatively, if using an application framework such as Spring Boot or Micronaut, use the built-in configuration mechanism which supports all three formats.

If it is necessary to support free-form configuration *in addition* to some number of expected items, add a `Map<String, Object>` attribute to the configuration class. When staging the Kafka configuration, apply the free-form properties first, then apply the expected ones. When applying the expected properties, check if the staging configuration already has a value set — if it does, throw a runtime exception. While this doesn't protect against misspelt property names in the free-form section, it will at least ensure that the expected configuration takes precedence.

For those cases when there is an absolute requirement for the property names to be correct before initialisation, one can take their validation a step further: First, the `java.lang.reflect` package can be used to scan the static constants in `CommonClientConfigs`, `ProducerConfig` or `ConsumerConfig`, then have those constants cross-checked against the user-supplied property names. The validation method will bail with a runtime exception if a given property name could not be resolved among the scanned constants.

The rest of this section will focus on the uncompromising, fully type-safe case. Expect a bit of coding. All sample code is provided at github.com/ekoutanov/effectivekafka[25], in the `src/main/java/effectivekafka/types` directory.

There are two classes in this example. The first defines a self-validating structure for containing producer client configuration. It has room for both expected properties and a free-form set of custom properties. The listing for `TypesafeProducerConfig` follows.

[25] https://github.com/ekoutanov/effectivekafka/tree/master/src/main/java/effectivekafka/typesafeproducer

```
import static java.util.function.Predicate.*;

import java.lang.reflect.*;
import java.util.*;
import java.util.stream.*;

import org.apache.kafka.clients.*;
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.config.*;
import org.apache.kafka.common.serialization.*;

public final class TypesafeProducerConfig {
  public static final class UnsupportedPropertyException
      extends RuntimeException {
    private static final long serialVersionUID = 1L;

    private UnsupportedPropertyException(String s) { super(s); }
  }

  public static final class ConflictingPropertyException
      extends RuntimeException {
    private static final long serialVersionUID = 1L;

    private ConflictingPropertyException(String s) { super(s); }
  }

  private String bootstrapServers;

  private Class<? extends Serializer<?>> keySerializerClass;

  private Class<? extends Serializer<?>> valueSerializerClass;

  private final Map<String, Object> customEntries = new HashMap<>();

  public TypesafeProducerConfig
      withBootstrapServers(String bootstrapServers) {
    this.bootstrapServers = bootstrapServers;
    return this;
  }

  public TypesafeProducerConfig withKeySerializerClass(
        Class<? extends Serializer<?>> keySerializerClass) {
    this.keySerializerClass = keySerializerClass;
```

```
    return this;
  }

  public TypesafeProducerConfig withValueSerializerClass(
        Class<? extends Serializer<?>> valueSerializerClass) {
    this.valueSerializerClass = valueSerializerClass;
    return this;
  }

  public TypesafeProducerConfig withCustomEntry(String propertyName,
                                                Object value) {
    Objects.requireNonNull(propertyName,
                           "Property name cannot be null");
    customEntries.put(propertyName, value);
    return this;
  }

  public Map<String, Object> mapify() {
    final var stagingConfig = new HashMap<String, Object>();
    if (! customEntries.isEmpty()) {
      final var supportedKeys =
          scanClassesForPropertyNames(SecurityConfig.class,
                                      SaslConfigs.class,
                                      ProducerConfig.class);
      final var unsupportedKey = customEntries.keySet()
          .stream()
          .filter(not(supportedKeys::contains))
          .findAny();

      if (unsupportedKey.isPresent()) {
        throw new UnsupportedPropertyException(
            "Unsupported property " + unsupportedKey.get());
      }

      stagingConfig.putAll(customEntries);
    }

    Objects.requireNonNull(bootstrapServers,
                           "Bootstrap servers not set");
    tryInsertEntry(stagingConfig,
                   ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
                   bootstrapServers);
    Objects.requireNonNull(keySerializerClass,
```

```
                          "Key serializer not set");
    tryInsertEntry(stagingConfig,
                   ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                   keySerializerClass.getName());
    Objects.requireNonNull(valueSerializerClass,
                          "Value serializer not set");
    tryInsertEntry(stagingConfig,
                   ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                   valueSerializerClass.getName());

    return stagingConfig;
  }

  private static void tryInsertEntry(Map<String, Object> staging,
                                     String key,
                                     Object value) {
    staging.compute(key, (__key, existingValue) -> {
      if (existingValue == null) {
        return value;
      } else {
        throw new ConflictingPropertyException("Property "
            + key + " conflicts with an expected property");
      }
    });
  }

  private static Set<String>
      scanClassesForPropertyNames(Class<?>... classes) {
    return Arrays.stream(classes)
        .map(Class::getFields)
        .flatMap(Arrays::stream)
        .filter(TypesafeProducerConfig::isFieldConstant)
        .filter(TypesafeProducerConfig::isFieldStringType)
        .filter(not(TypesafeProducerConfig::isFieldDoc))
        .map(TypesafeProducerConfig::retrieveField)
        .collect(Collectors.toSet());
  }

  private static boolean isFieldConstant(Field field) {
    return Modifier.isFinal(field.getModifiers())
        && Modifier.isStatic(field.getModifiers());
  }
```

```
  private static boolean isFieldStringType(Field field) {
    return field.getType().equals(String.class);
  }

  private static boolean isFieldDoc(Field field) {
    return field.getName().endsWith("_DOC");
  }

  private static String retrieveField(Field field) {
    try {
      return (String) field.get(null);
    } catch (IllegalArgumentException | IllegalAccessException e) {
      throw new RuntimeException(e);
    }
  }
}
```

The `TypesafeProducerConfig` class defines a pair of public nested runtime exceptions — `UnsupportedPropertyExcept` and `ConflictingPropertyException`. The former will be thrown if the user provides a custom property with an unsupported name, trapping those pesky typos. The latter occurs when the custom property conflicts with an expected property. Both are conditions that we ideally would prefer to avoid before initialising the producer client.

Next, we declare our private attributes. This example expects three properties — `bootstrapServers`, `keySerializerClass`, and `valueSerializerClass`. For the serializers, we have taken the extra step of restricting their type to `java.lang.Class<? extends Serializer<?>>`, ensuring that only valid serializer implementations may be assigned. The `customEntries` attribute is responsible for accumulating any free-form entries, supplied in addition to the expected properties.

The `mapify()` method is responsible for converting the stored values into a form suitable for passing to a `KafkaProducer` constructor. It starts by checking if `customEntries` has at least one entry in it. If so, it invokes the `scanClassesForPropertyNames()` method, passing it the class definitions of `ProducerConfig` as well as some common security-related configuration classes. The result will be a set of strings harvested from those classes using reflection. The `ProducerConfig` class already imports the necessary constants from `CommonClientConfigs`, relieving us from having to scan `CommonClientConfigs` explicitly.

The actual implementation of `scanClassesForPropertyNames()` should hopefully be straightforward, requiring basic knowledge of Java 8 and the `java.util.stream` API. Essentially, it enumerates over the `public` fields, filtering those that happen to be constants (having `final` and `static` modifiers), are of a `java.lang.String` type, and are not suffixed with the string `_DOC`. This set of filters should round up all supported property names. The filtered fields are retrieved and packed into a `java.util.Set`. For convenience, the code listing of `scanClassesForPropertyNames()` is repeated below.

```
private static Set<String>
    scanClassesForPropertyNames(Class<?>... classes) {
  return Arrays.stream(classes)
      .map(Class::getFields)
      .flatMap(Arrays::stream)
      .filter(TypesafeProducerConfig::isFieldConstant)
      .filter(TypesafeProducerConfig::isFieldStringType)
      .filter(not(TypesafeProducerConfig::isFieldDoc))
      .map(TypesafeProducerConfig::retrieveField)
      .collect(Collectors.toSet());
}
```

The filter in `scanClassesForPropertyNames()` may also inadvertently include other string constants in the given `classes` array that happen to match its predicates. Kafka's maintainers haven't consistently differentiated between supported property names and other constants. The majority use the `_CONFIG` suffix to indicate a supported name; however, `ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION` has strayed from this convention. Short of specifying an exclusion filter that blacklists known stray constants, there is little we can do about this. A blacklist would create a long-term maintenance headache and is hardly worth the effort. The likelihood of a misspelt user-supplied property name colliding with a stray constant is negligible.

Let's now use our `TypesafeProducerConfig` to configure an actual client:

```
final var config = new TypesafeProducerConfig()
    .withBootstrapServers("localhost:9092")
    .withKeySerializerClass(StringSerializer.class)
    .withValueSerializerClass(StringSerializer.class)
    .withCustomEntry(
        ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 1);

try (var producer = new KafkaProducer<>(config.mapify())) {
  // do something with producer
}
```

The use of `TypesafeProducerConfig` follows a fluent style of method chaining; the code is compact but readable. But crucially, it is now bullet-proof. You can see it for yourself: feed an invalid property name into `withCustomEntry()` and watch `config.mapify()` bail with a runtime exception, avoiding the initialisation of `KafkaProducer`.

We can also populate a `TypesafeProducerConfig` object from a JSON or YAML configuration file using a parser such as Jackson, available at github.com/FasterXML/jackson[26]. Jackson supports a

[26] https://github.com/FasterXML/jackson

wide range of formats, including JSON, YAML, TOML, and even plain `.properties` files. To map the parsed configuration document to a `TypesafeProducerConfig` instance, Jackson requires either the addition of Jackson-specific annotations to our class or defining a custom deserializer.

Alternatively, if using an application framework such as Spring Boot or Micronaut, we can wire a `TypesafeProducerConfig` object into the framework's native configuration mechanism. This typically requires the addition of setter methods to make the encapsulated attributes writable by the framework. The design of `TypesafeProducerConfig` defers all validation until the `mapify()` method is invoked, thereby remaining agnostic of how its attributes are populated.

The examples in this book are intentionally decoupled from application frameworks. The intention is to demonstrate the correct and practical uses of Kafka, using the simplest and most succinct examples. These examples are intended to be run as standalone applications using any IDE.

You might have noticed that the example still used a constant to specify the custom property name, in spite of having a reliable mechanism for detecting misspelt names early, before client initialisation. Using constants traps errors at compile-time, which is the holy grail of building robust software.

While the `TypesafeProducerConfig` example solves the validation problem in its current guise, the presented solution is not very reusable. It would take a fair amount of copying and pasting of code to apply this pattern to different configuration classes, even if the variations are minor. With a modicum of refactoring, we can turn the underlying ideas into a generalised model for configuration management.

If you are interested, take a look at github.com/ekoutanov/effectivekafka[a]. The `src/main/java/effectivekafka/config` directory contains an example of how this can be achieved. The `AbstractClientConfig` class serves as an abstract base class for a user-defined configuration class. The base class contains the `customEntries` attribute, as well as the validation logic for ensuring the correctness of property names. The deriving class is responsible for providing the expected attributes and the related validation logic. The `mapify()` method lives in the base class, and will invoke the subclass to harvest its expected configuration properties.

[a] https://github.com/ekoutanov/effectivekafka/tree/master/src/main/java/effectivekafka/config

---

This chapter identified the challenges with configuring Kafka clients — namely, Kafka's permissive stance on validating the user-supplied configuration and treating configuration entries as a typeless map of string-based keys to arbitrary values.

The problem of validating configuration for the general case can be solved by creating an intermediate configuration class. This class houses the expected configuration items as type-safe attributes,

as well as free-form configuration, which is validated using reflection. A configuration class acts as an intermediate placeholder for configuration entries, enabling the application to proactively validate the client configuration before instantiating the client, in some cases using nothing more than compile-time type safety.

Chapter 12: Batching and Compression

Chapter 10: Client Configuration mostly attended to the aspects of client configuration related to functionality and safety, deliberately sidestepping any serious discussions on performance. The intent of this chapter is to focus on one specific area of performance optimisation — batching and compression. The two are related, collectively bearing a significant impact on the performance of an event streaming system.

Comparing disk and network I/O

Kafka utilises a segmented, *append-only log*, largely limiting itself to *sequential I/O* for both reads and writes, which is fast across a wide variety of storage media. There is a wide misconception that disks are slow; however, the performance of storage media (particularly rotating media) is greatly dependent on access patterns. The performance of random I/O on a typical 7,200 RPM SATA disk is between three and four orders of magnitude slower than sequential I/O. Furthermore, a modern operating system provides read-ahead and write-behind techniques that prefetch data in large block multiples and group smaller logical writes into large physical writes. Because of this, the difference between sequential I/O and random I/O is still evident in flash and other forms of solid-state non-volatile media, although the effects are less dramatic compared to rotating media.

Sequential I/O is comparable to the peak performance of network I/O. Furthermore, disk I/O is local to the host, whereas network I/O is shared. In practice, this means that a well-designed log-structured persistence layer will keep up with the network traffic. In fact, often the bottleneck with Kafka's performance isn't the disk, but the network. Stated bluntly, the network fabric will run out of steam before the broker.

Producer record batching

To counteract the limitations of the network, Kafka clients will batch multiple records together before sending them over the network. This is independent of, and in addition to, the low-level batching provided by the OS at the TCP socket layer. Batching of records amortises the overhead of the network round-trip, using larger packets and improving bandwidth efficiency.

Batching in Kafka can act end-to-end. The producer client uses an accumulator buffer for staging records prior to forwarding them to the leader broker. Once the records are batched, the broker will (in most cases) persist them as-is, without unpacking the batch or performing any intermediate

manipulations of the stored records. This is carried through to the consumer. When polling for records, the consumer is served batches of records by the broker — the same batches that were originally published. There are cases where a batch is not end-to-end; for example, when a broker is instructed to apply a compression scheme that differs from the producer. More on that later.

Being a largely end-to-end concern, batching is controlled by the producer. The `batch.size` and `linger.ms` properties collectively limit the extent to which the producer will attempt to batch queued records in order to maximise the outgoing transmission efficiency. The default values of `batch.size` and `linger.ms` are `16384` (number of bytes) and `0` (milliseconds), respectively.

The producer combines any records that arrive between request transmissions into a single batched request. Normally this only occurs under load, when records arrive faster than they can be transmitted. In some circumstances, the client may want to reduce the number of requests even under moderate load. The `linger.ms` setting accomplishes this by adding a small amount of artificial delay — rather than immediately sending a record the moment it is enqueued, the producer will wait for up to a set delay to allow other records to accumulate in a batch, maximising the amount of data that can be transmitted in one go. Although records may be allowed to linger for up to the duration specified in `linger.ms`, the `batch.size` property will have an overriding effect, dispatching the batch once it reaches the set maximum size.

While the `linger.ms` property only comes into the picture when the producer is lightly loaded, the `batch.size` property is continually in effect, acting as the overarching and unremitting limiter of the staged batch.

> The `linger.ms` setting is often likened to the *Nagle's Algorithm* in TCP, as they both aim to improve the efficiency of network transmission by combining small and intermittent outgoing messages and sending them at once. The comparison was originally made by the Apache Kafka design team. It has since found its way into the official documentation and appears to reverberate strongly within the user community.
>
> While the similarities are superficial, there are two crucial differences. Firstly, Nagle's Algorithm does not indiscriminately buffer data on the basis of its apparent intermittence. It only takes effect when there is *at least one outstanding packet* that is yet to be acknowledged by the receiver. Secondly, Nagle's Algorithm does not impose an artificial time limit on the delay. Data will be buffered until a full packet is formed, or the `ACK` for the previous packet is received, whichever occurs first.
>
> Combined, the two instruments make the algorithm self-regulating; rather than relying on arbitrary, user-defined linger times, the algorithm buffers data based on the network's observed performance. Free-flowing networks with frequent `ACK`s result in lighter buffering and reduced transmission latency. As congestion increases, the buffering becomes more pronounced, improving transmission efficiency at the expense of latency, and helps avoid a congestive collapse.

Congestive collapse is a phenomenon that occurs when a network is overwhelmed by a high packet rate, usually at known choke points, leading to increased failures (packet loss and timeouts) and a corresponding escalation of retries. This creates a perpetuating feedback loop that degenerates to a parasitic stable state where the traffic demand is high, but there is little useful throughput available.

The batching algorithm used by Kafka is crude by comparison. It requires careful tuning and imposes a fixed penalty on intermittent publishing patterns regardless of the network's performance or the observed latency. This does not necessarily render it ineffective. On the contrary, the batching algorithm can be very effective over high-latency networks. Its main issue is the lack of adaptability, making it suboptimal in dynamic network climates or in the face of varying cluster performance — both factors affecting publishing latency. Kafka places the onus of tuning the algorithm parameters on the user, requiring careful and extensive experimentation to empirically arrive at the optimal set of parameters for a given network and cluster profile. These parameters must also be periodically revised to ensure their continued viability.

Even with the default `linger.ms` value of `0`, the producer will still allow for some buffering due to the asynchronous nature of the transmission: calling `send()` serializes the record, assigns a partition number, and places the serialized contents into the accumulator buffer, but does not transmit it. The actual communications are handled by a background I/O thread. So if `send()` is called multiple times in rapid succession, at least some of these records will likely be batched.

Due to the diminishing returns exhibited with larger batch sizes and the lack of self-regulation in Kafka's batching algorithm, it may be prudent to err on the side of smaller values of `linger.ms` initially — from zero to several milliseconds. The non-functional requirements of the overall system should also specify the tolerance for latency, which ought to be honoured over any gains in network efficiency or storage savings on the brokers. The extent of batching may be increased further if the network is becoming a genuine bottleneck; however, such changes should be temporary — the focus should be on addressing the root cause.

Compression

The effectiveness of batching increases substantially when complemented by record compression. As compression operates on an entire batch, the resulting compression ratios increase with the batch size. The effects of compression are particularly pronounced when using text-based encodings such as JSON, where the records exhibit low information entropy. For JSON, compression ratios ranging from 5x to 7x are not unusual, which makes enabling compression a no-brainer. Furthermore, record batching and compression are largely done as a client-side operation, which transfers the load onto the client and has a positive effect not only on the network bandwidth, but also on the brokers' disk I/O and storage utilisation.

The biggest gains will be felt when starting from small batches. As the batch size increases, the laws of diminishing returns take effect — an increase in the batch size will yield a proportionally smaller gain in compression ratio. The incremental reduction in the number of packets used, and hence the transmission efficiency, will also be less noticeable with higher batch sizes.

The `compression.type` client configuration property controls the algorithm that the producer will use to compress record batches before forwarding them on to the partition leaders. The valid values are:

- `none`: Compression is disabled. This is the default setting.
- `gzip`: Use the *GNU Gzip* algorithm — released in 1992 as a free substitute for the proprietary `compress` program used by early UNIX systems.
- `snappy`: Use Google's *Snappy* compression format — optimised for throughput at the expense of compression ratios.
- `lz4`: Use the *LZ4* algorithm — also optimised for throughput, most notably for the speed of decompression.
- `zstd`: Use Facebook's *ZStandard* — a newer algorithm introduced in Kafka 2.1.0, intended to achieve an effective balance between throughput and compression ratios.

Kafka compression applies to an entire record batch, which as we know *can be* end-to-end — depending on the settings on the broker. Specifically, when the broker is configured to accept the producer's preferred compression scheme, it will not interfere with the contents of the batch. The batch flows from the producer to the broker, is persisted across multiple replicas, and is eventually served to one or more consumers — all as a single, indivisible chunk. The broker simply acts as a relaying party — it does not decompress and re-compress the record as part of its role. Each chunk has a header that notes the algorithm that was used during compression, allowing the consumers to apply the same when unpacking the chunk.

The `compression.type` broker property applies to all topics by default, overriding the producer property. In addition, it is possible to configure individual topics by using the `kafka-configs.sh` CLI to specify a dynamic configuration for the `topics` entity type.

On the flip side, end-to-end compression has a subtle drawback, which can catch out unsuspecting users. Because the broker is unable to mediate the interchange format, the result is a latent coupling between the producer's capabilities and that of the consumer clients. Although Kafka strives to maintain binary protocol compatibility between minor releases, this promise does not cover end-to-end contracts, such as compression and checksumming. As such, the producer application must use a compression format this is compatible with the oldest consumer version.

The introduction of ZStandard is a good example of a breaking change, and may be considered as a 'gotcha' depending on your client ecosystem. When operating a mixture 2.1.0+ and pre-2.1.0 consumers, the use of `compression.type=zstd` on a producer will render the records unreadable for the older clients, resulting in an `UNSUPPORTED_COMPRESSION_TYPE` error. The correct way to enable ZStandard is to upgrade all consumers first, and only when the last pre-2.1.0 consumer has been retired, allow the use of `compression.type=zstd`. Alternatively, one can enable re-compression on the broker to maintain compatibility with older clients; however, this leads to increased resource utilisation on the broker.

End-to-end compression has a profoundly positive impact on performance. Compression is a processor-intensive operation and consumes additional memory, particularly during the encoding phase. (Comparatively, decoding a compressed stream is cheaper; the difference may be an order of magnitude in extreme cases.) By eliminating the broker from the equation, the cost of compression is absorbed by the clients. The distribution of load to the periphery dovetails into Kafka's broader scaling strategy — it is typically much easier to scale the clients than the broker, at least for well-architected, stateless applications.

Kafka additionally offers compression at the broker level, for those scenarios where it is necessary to change the compression scheme. This is controlled by the `compression.type` broker property. Its default value is `producer`, meaning that the broker will revert to the producer-assigned compression scheme; in other words, the broker will not meddle the batch. Alternatively, the value of `compression.type` may be set to one of the supported compression schemes (as per the producer-side property). The only difference is when disabling compression: the producer property accepts `none`, whereas the broker property accepts `uncompressed`.

While broker-level configuration can provide for more fine-grained control of the compression scheme, its main drawback is the increased CPU utilisation and the forfeiting of the *zero-copy* optimisation, as the batches are no longer end-to-end.

Zero-copy describes computer operations in which the CPU does not perform the task of copying data from one memory area to another. In a typical I/O scenario, the transfer of data from a network socket to a storage device occurs without the involvement of the CPU and with a reduced number of context switches between the kernel and user mode.

The use of compression has no functional effect on the system. Its effect on bandwidth utilisation, disk space utilisation, and broker I/O are, in some cases, astounding. As a performance optimisation, and depending on the type of data transmitted, the effect of compression can be so profound that it thwarts any potential philosophical debates over the viability of 'premature' optimisation.

Compression algorithms achieve a reduction in the encoded size relative to the uncompressed original by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the uncompressed data stream. For compression to be effective, the data must contain a large amount of repetition and be of sufficient size so as to warrant any structural

overheads, such as the introduction of a dictionary. Text formats such as JSON tend to be highly compressible as they are verbose by nature and contain repeated character sequences that carry no informational content. They also fail to take advantage of the full eight bits that each byte can theoretically accommodate, instead representing characters with seven of the lower order bits. Binary encodings tend to exhibit more informational density, but may still be highly compressible, depending on their internal structure. Binary streams containing digital media — for example, images, audio or video — are high-entropy sources and are virtually incompressible.

Information entropy is the measure of the informational content conveyed by an element in a stream. It is determined by the likelihood of predicting the value of an element in a stream based on the observations of prior elements, with the resulting score varying between zero (no entropy, perfectly predictable) and one (highest entropy, completely unpredictable). In the context of a data record, an element might be an individual bit or a byte in the record. The easier it is to predict the value of the next byte, the less new information it carries. As an example of a low entropy source, consider a formatted JSON document. Having observed a newline character, it is extremely likely that the next element is a whitespace character, with the next likely candidate being a double quote, followed in relative likelihood by a closing brace. Compare this to a truly randomly sequence of bytes. The likelihood of predicting the next byte is equivalent to chance; this is an example of maximum entropy. Compressing this type of data will only lead to an increase in the output size as the overheads of the compression algorithm will be added into the output, not offset by any gains in entropy.

Without delving into a harrowing analysis of the different compression schemes, the recommendation is to always enable compression for text encodings as well as binary data (unless the latter is known to contain a high information entropy payload). In some cases, the baseline impact of enabling compression may be so substantial that the choice of the compression algorithm hardly matters. In other cases, the choice of the algorithm may materially impact the overall performance, and a more careful selection is warranted.

As a rule of thumb, use LZ4 when dealing with legacy consumers, transmitting over a network that offers capacity in excess of your peak uncompressed data needs. In other words, the network is able to sustain your traffic flow even without compression. If the network has been identified as the bottleneck, consider switching to Gzip and also increasing `batch.size` and `linger.ms` to increase the size of transmitted batches to maximise the effectiveness of compression at the expense of latency.

If all consumers are at a version equal to or greater than 2.1.0, your choices are basically LZ4 and ZStandard. Use LZ4 for low-overhead compression. When the network becomes the bottleneck, consider ZStandard, as it is able to achieve similar compression ratios to Gzip at a fraction of the compression and decompression time. You may also need to increase `batch.size` and `linger.ms` to maximise compression effectiveness.

The guidelines above should not be taken to mean that LZ4 always outperforms Snappy or that

ZStandard is should always be preferred over Gzip. When undertaking serious performance tuning, you should carefully consider the shape of your data and conduct studies using synthetic records that are representative of the real thing, or better still, using historical data if this is an option. When benchmarking the different compression schemes, you should also measure the CPU and memory utilisation of the producer and consumer clients, comparing these to the baseline case (when compression is disabled). In some cases you may find that Snappy or Gzip indeed offer a better compromise. The guidelines presented here should be used as the starting point, particularly when one's copious free time is prioritised towards dealing with the matters of building software, over conducting large-scale performance trials.

This chapter has given the reader an insight into Kafka's performance 'secret sauce' — namely, the use of log-structured persistence to limit access patterns to sequential reads and writes. The implication: a blazingly fast disk I/O subsystem that can outperform the network fabric, requiring further client-side optimisations to bring the two into parity.

We looked at two controls available on the producer client — batching and compression. The potential performance impacts of these controls are significant, particularly in the areas of throughput and latency. Getting them right could entail significant gains with relatively little effort.

Chapter 13: Replication and Acknowledgements

Fundamentally, Apache Kafka is a distributed log-centric data store. Data is written across multiple nodes in a cluster and may be subject to a range of contingencies — disk failures, intermittent timeouts, process crashes, and network partitions. How Kafka behaves in the face of a contingency and the effect this has on the published data should be of material concern to the designer of an event-driven system.

This chapter explores one of the more nuanced features of Kafka — its replication protocol.

Replication basics

The deliberate decisions made during the design of Kafka ensure that data written to the cluster will be both *durable* and *available* — meaning that it will survive failures of broker nodes and will be accessible to clients. The replication protocol is the specific mechanism by which this is achieved.

As it was stated in Chapter 3: Architecture and Core Concepts, the fundamental unit of streaming in Kafka is a partition. For all intents and purposes, a partition is a replicated log. The basic premise of a replicated log is straightforward: data is written to multiple replicas so that the *failure of one replica does not entail the loss of data*. Furthermore, replicas must *agree* among themselves with respect to the contents of the replicated log — reflecting on their local facsimile of the log. It would be unacceptable for two (or more replicas) to differ in their contents in some conflicting manner, as this would lead to data corruption. Broadly speaking, this notional agreement among the replicas is referred to as *distributed consensus*, and is one of the basic challenges faced by the designers of distributed systems.

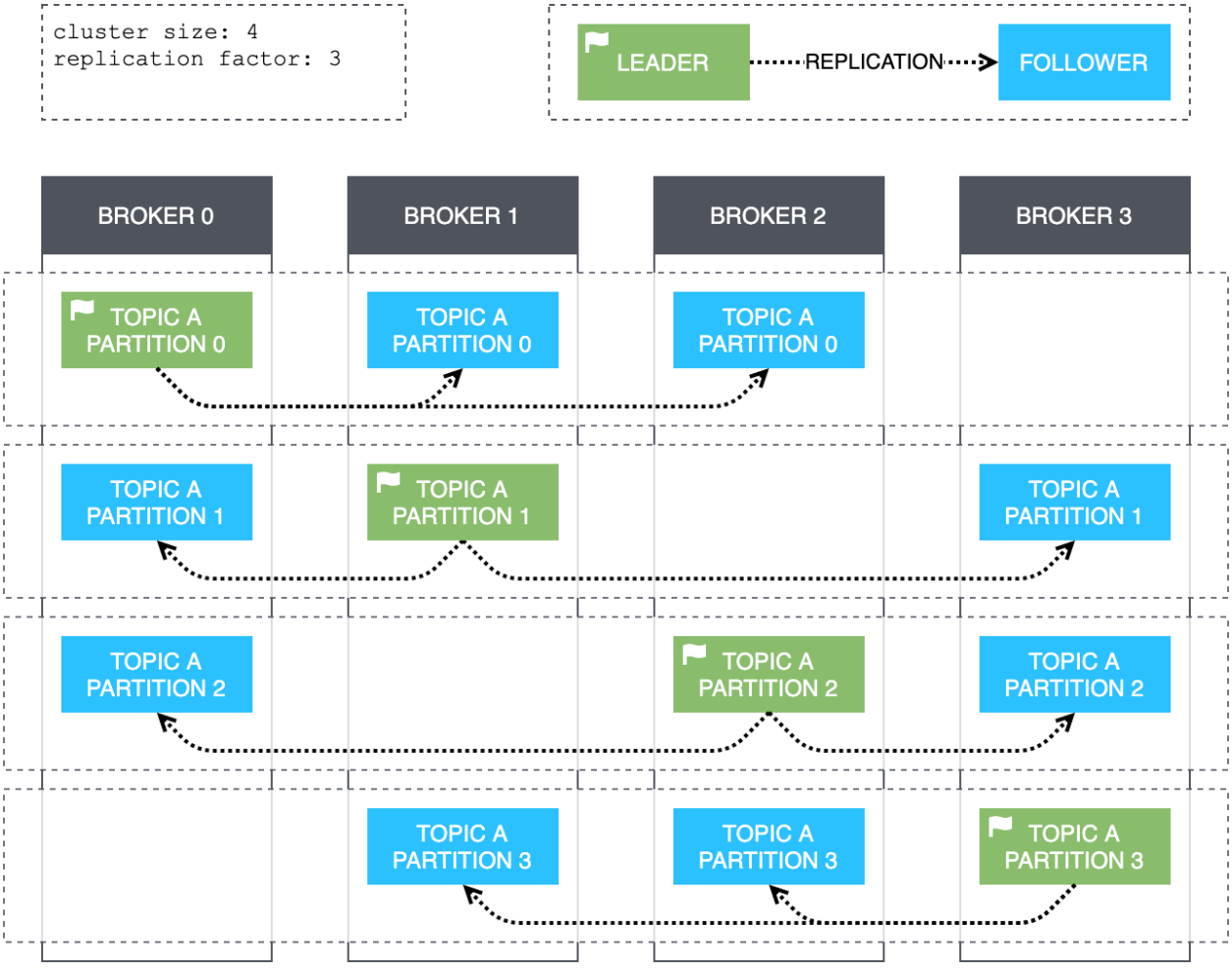
There are several approaches to implementing a distributed log; the one taken by Kafka follows a leader-follower model — a single leader is assigned by the cluster coordinator to take absolute mastership of the partition, with zero or more followers that tail the data written by the leader in near real-time, progressively building their own identical copies of the log. Putting it another way, replication in Kafka is *asynchronous*: replicas lag behind the leader, converging on its state when the traffic flow from the leader quiesces. Consensus is formed by ensuring that only one party administers changes to the log; all other parties implicitly agree by unconditionally replicating all changes from the leader, achieving *sequential consistency*. Under this consistency model, replicas can only vary in a contiguous segment comprising the last few records in the log — they cannot have gaps, nor can two replicas house different records at the same position in the log. This model vastly simplifies the consensus protocol, but some level of agreement is required nonetheless, because

sequential consistency is insufficient on its own. The protocol must ensure that records are durably persisted, which implies that certain aspects of the protocol must be synchronous.

In Kafka's parlance, both the leader and the follower roles are collectively referred to as *replicas*. The number of replicas is configured at the topic level, and is known as the topic's *replication factor*. During the exercises in Chapter 5: Getting Started, we created topics with a replication factor of one. There was no other choice then, as our test cluster comprised a single broker node. In practice, production clusters will comprise multiple nodes — three is often the minimum, although larger clusters are common.

The minimum permitted replication factor is one — offering no redundancy. Increasing the replication factor to two provides for a single follower replica, but will prevent further modifications to the data if one of the replicas fails. This configuration provides durability, but as for availability — this is only provided in the read aspect, as writes are not highly available. A replication factor of three provides both read and write availability, providing certain other conditions are met. Naturally, the replication factor cannot exceed the size of the cluster.

A naive replication model with a replication factor of three is depicted below. This is a simplification of what actually happens in Kafka, but it is nonetheless useful in visualising the relationship between leader and follower replicas. It will be followed shortly with a more complete model.



Topic replication — simplified model

Broker nodes are largely identical in every way; each node competes for the mastership of partition data on equal footing with its peers. Given the symmetric nature of the cluster, Kafka designates a single node – the *cluster controller* – for managing the partition assignments within the cluster. Partition leadership is apportioned approximately evenly among the brokers in a cluster when a topic is first created. Kafka allows the cluster to scale to a greater number of topics and partitions by adding more broker nodes; however, changes to the cluster size require explicit rebalancing of replicas on the operator's behalf. More on that later.

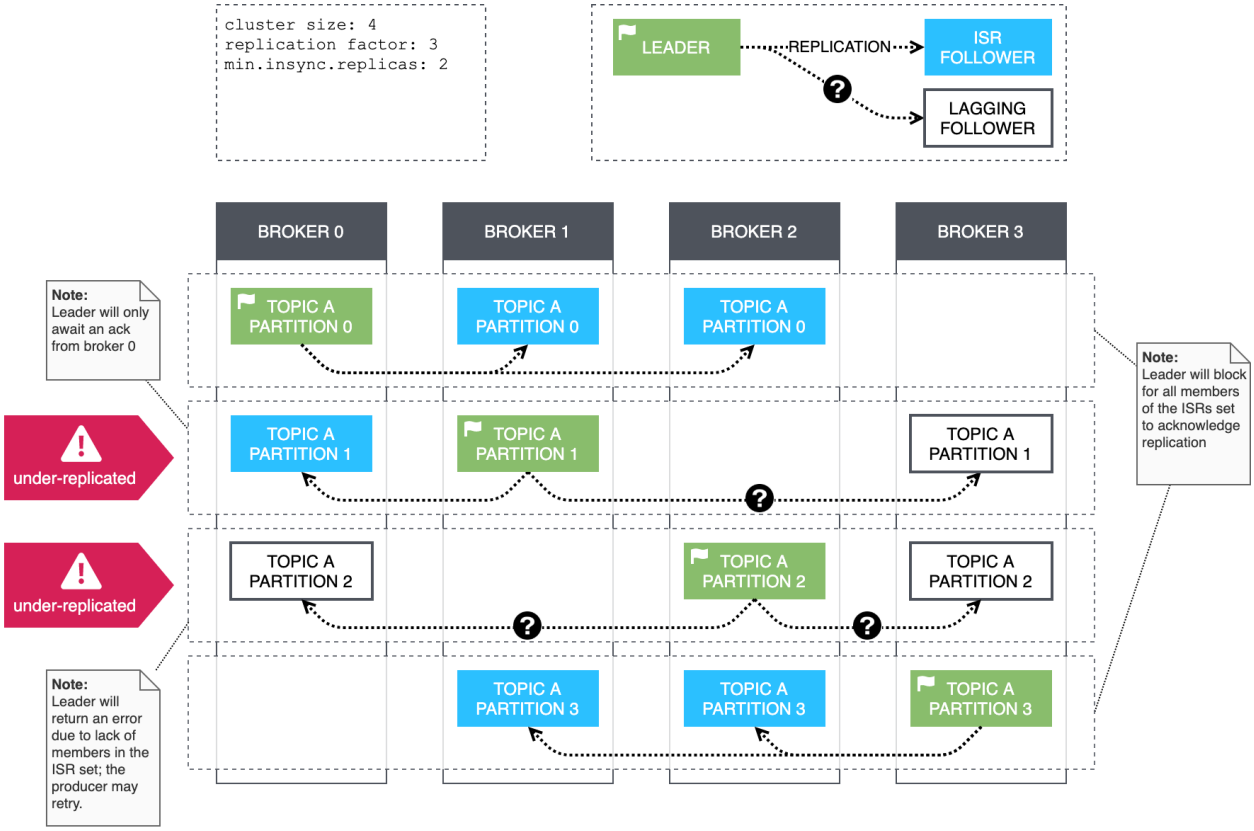
The main challenge with naively replicating data from a leader to a follower in the manner above, is that in order to guarantee durability, the leader must wait for the all replicas to acknowledge the write — before reporting to the producer that the write has been replicated to the degree implied by the replication factor. This makes the replication protocol sensitive to slow replicas, as the acknowledgement times are dependent on the slowest replicas. A single replica that is experiencing a period of degraded performance will affect all durable writes to the partition in question. Also, waiting for all replicas is not strictly necessary to achieving consensus in a replicated log.

Note, we use the term 'durable write' to refer to the persistence of those records where the producer has requested the highest level of durability. This is not necessarily the case with all writes; the producer can dictate the level of durability by specifying the number of required acknowledgements.

To lighten the burden of slow replicas, Kafka introduces the concept of *In-Sync Replicas*, abbreviated to *ISR*. This is a dynamically allocated set of replicas that can demonstrably keep pace with the leader. The leader is also included in the ISR. Normally, this implies that a replica is trailing the leader within some bounded window of time. (The time window being the difference between the timestamp of the record at the leader's log end offset and that of the follower.) Slow replicas are automatically removed from the ISR set, and as such, the cardinality of the ISR set may be lower than the replication factor. A partition that has had at least one replica removed from the ISR is said to be *under-replicated*. The responsibility of evaluating the performance of the ISR and maintaining the state of the ISR falls on the partition leader, which must also persist a copy of the ISR to ZooKeeper upon every change. This ensures that, should the leader fail, any of the followers can reconstruct the state of the ISR — enabling them to take over the leadership of the topic.

Instead of requiring the leader to garner acknowledgements from all follower replicas as in the earlier example, a durable write only requires that acknowledgements are received from those replicas in the ISR. The lower bound on the size of the ISR is specified by the `min.insync.replicas` configuration property on the broker. This property is complementary to the `default.replication.factor` property, being the defaults that will be applied to all topics. When creating a topic, its custodian may set an alternate replication factor and override the `min.insync.replicas` property as required. The producer client has no say in the replication factor or the minimum size of the in-sync replica set — it can only stipulate whether a write should be durable or not. Naturally, the consumer client has no say in any matter regarding durability.

When a partition leader is asserting a durability guarantee, it must ensure that all replicas in the ISR have acknowledged the write, before responding to the producer. Because the ISR automatically excludes underperforming replicas, the performance of the replication protocol is minimally impacted by a stalled or lagging replica. To be specific, a deteriorated replica will still affect the replication performance for as long as it is deemed a member of the ISR, but eventually it will be removed from the ISR — at that point and thereafter it will have no impact on performance. Prior to rejoining the ISR, a replica must catch up to the leader. A replication scheme based on ISR has been depicted below.



Topic replication — model with ISR

In the diagram above, the topic has been configured with a replication factor of three and a minimum ISR size of 2. For those partitions where there are three replicas in the ISR — a confirmation from both followers is necessary before a durable write can be acknowledged by the leader. For those partitions where the ISR has been reduced to two, the leader will wait for the remaining in-sync replica other than itself, before acknowledging the write. Finally, when the ISR is completely depleted, the leader will communicate an error back to the initiating publisher, which in turn, may reattempt to publish the record.

> Comparing this model to other distributed consensus protocols, such as ZAB, Raft, and Paxos: one no longer requires majority agreement among the cohorts before accepting a log write and thereby deeming it stable. Provided that there is consensus on the membership of the ISR, and all members of the ISR are synchronised with the leader, then the ISR does not need to constitute the majority of the replicas. Conceivably, we could have 100 replicas, with only two in the ISR, and still achieve consensus.
>
> At first glance, it may appear that Kafka's replication protocol has accomplished a feat that eluded distributed systems researches for several decades. This is not the case. The trick is in the consensus on the ISR membership state, which is backed by ZooKeeper. ZooKeeper's ZAB protocol provides the underlying primitives for Kafka to build upon. So although Kafka does not require a majority vote for log replication, it does require a majority vote for making updates to the ISR state. Putting

> it another way, Kafka layers its replication protocol on top of ZAB, forming a hybrid of the two.

The default value of the `min.insync.replicas` property is `1`, which implies that a durable write only extends to the leader replica, which is hardly durable. At minimum, `min.insync.replicas` should be set to `2`, ensuring that a write is reflected on the leader as well as at least one follower. This property can be set for all topics, as well as for individual topics. Instructions for targetting specific topics have been covered in Chapter 9: Broker Configuration.

The maximum tolerable replication lag is configured via the `replica.lag.time.max.ms` broker property, which defaults to `10000` (ten seconds). If a follower hasn't sent any fetch requests or hasn't consumed up to the leader's log end offset within this time frame, it will be summarily dismissed from the ISR. This setting can be applied to all topics, or selectively to individual topics.

Prior to Kafka 0.8.3.0, the maximum tolerable replication lag was configured via the `replica.lag.max.messages` property. The replication protocol used to consider the number of records that a follower was trailing by, to determine whether it should be in the ISR. Replicas could easily be knocked out of the ISR during sudden bursts of traffic, only to rejoin shortly afterwards. Conversely, low-volume topics would take a long time to detect an out-of-sync replica. As part of KIP-16[27], the protocol has since evolved to only consider the time delay between the latest record on the leader and that of each follower. This made it easier to tune the protocol, as it was less susceptible to flutter during traffic bursts. It also made it easier to set meaningful values, as it is more natural to think of lag in terms of time, rather than in terms of arbitrary records.

Leader election

Only members of the ISR are eligible for leader election. Recall, Kafka's replication protocol is generally asynchronous and a replica in the ISR is not guaranteed to have all records that were written by the outgoing leader, only those records that were confirmed by the leader as having been durably persisted. In other words, the protocol is synchronous only with respect to the durable writes, but not necessarily all writes. It is conceivable then, that some in-sync replicas will have more records than others, while preserving sequential consistency with the leader. When selecting the new leader, Kafka will favour the follower with the highest log end offset, recovering as much of the unacknowledged data as possible.

Kafka's guarantee with respect to durability is predicated on at least one fully-synchronised replica remaining intact. Remember, an in-sync replica will contain all durable writes; the `min.insync.replicas` states the minimum number of replicas that will be fully-synchronised at any given time. When

[27] https://cwiki.apache.org/confluence/display/KAFKA/KIP-16+-+Automated+Replica+Lag+Tuning

suitably sized and appropriately configured, a cluster should tolerate the failure of a bounded number of replicas. But what happens when we've gone over that number?

At this point, Kafka essentially provides two options: either wait until an in-sync replica is restored or perform *unclean leader election*. The latter is enabled by setting `unclean.leader.election.enable` to `true` (it is `false` by default). Unclean leader election allows replicas that were not in the ISR at the time of failure to take over partition leadership, trading consistency for availability.

To maintain consistency in the face of multiple replica failures, one should set the replication factor higher than the minimum recommended value of three, and boost the `min.insync.replicas` value accordingly. This increases the likelihood of a surviving replica being fully-synchronised with the leader. One could further boost `min.insync.replicas` to equate to the replication factor, thereby ensuring that every replica is fit to act as a leader. The downside of this approach is the loss of availability in the write aspect: should a replica fail, consistency will be preserved, but no further writes to the partition will be allowed for having insufficient replicas in the ISR set. It is also less performant, negating the main purpose of an ISR — to reduce the performance impact of slow replicas.

To be clear, the consistency-availability tradeoffs alluded to above are not unique to Kafka's replication protocol, affecting every distributed consensus protocol. Given a constant number of replicas, one can increase the number of replicas that must agree on a write, thereby increasing the likelihood that at least one complete replica survives — but in doing so, impede the system's ability to make progress with a reduced replica set. Conversely, reducing the number of voting replicas will allow the system to tolerate a greater number of replica failures, but increases the likelihood of losing all complete replicas — leaving just the partial ones, or none at all. The only way to solve both problems simultaneously is to increase the replication factor, which increases the cost of the setup and impacts the performance of durable writes by requiring more acknowledgements. Whichever the approach, it has its drawbacks.

Setting the initial replication factor

The replication factor can be initially assigned when creating a topic using the Kafka Admin API. Alternatively, it can be set via the `--replication-factor` flag in the `kafka-topics.sh` CLI tool. Kafdrop also lets you set the replication factor when creating a topic. If unspecified, the replication factor for a newly created topic is sourced from the `default.replication.factor` broker configuration property, which is `1` by default.

Attempting to create a topic with a replication factor greater than the size of the cluster results in a `org.apache.kafka.common.errors.InvalidReplicationFactorException`.

Changing the replication factor

Once a replication factor has been set, either the Admin API or the CLI tool can be used to subsequently alter the replication factor. As it happens, this is not as straightforward as setting it initially; changing the replication factor is a semi-manual operation, which entails specifying a new set of replicas for each partition.

We need access to a multi-broker cluster, with at least two brokers, to practice changing the replication factor. If you happen to have a multi-broker cluster at your disposal — perfect. Otherwise, don't fret: you can easily spin one up using the following Docker Compose file. Remember to shut down any existing Kafka, ZooKeeper, and Kafdrop instances before spinning up a Compose stack, as it will conflict on port assignments.

```
version: "3.2"
services:
  zookeeper:
    image: bitnami/zookeeper:3
    ports:
      - 2181:2181
    environment:
      ALLOW_ANONYMOUS_LOGIN: "yes"
  kafka-0:
    image: bitnami/kafka:2
    ports:
      - 9092:9092
    environment:
      KAFKA_CFG_ZOOKEEPER_CONNECT: zookeeper:2181
      ALLOW_PLAINTEXT_LISTENER: "yes"
      KAFKA_LISTENERS: >-
          INTERNAL://:29092,EXTERNAL://:9092
      KAFKA_ADVERTISED_LISTENERS: >-
          INTERNAL://kafka-0:29092,EXTERNAL://localhost:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: >-
          INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: "INTERNAL"
    depends_on:
      - zookeeper
  kafka-1:
    image: bitnami/kafka:2
    ports:
      - 9093:9093
    environment:
      KAFKA_CFG_ZOOKEEPER_CONNECT: zookeeper:2181
```

```
      ALLOW_PLAINTEXT_LISTENER: "yes"
      KAFKA_LISTENERS: >-
          INTERNAL://:29092,EXTERNAL://:9093
      KAFKA_ADVERTISED_LISTENERS: >-
          INTERNAL://kafka-1:29092,EXTERNAL://localhost:9093
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: >-
          INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: "INTERNAL"
    depends_on:
      - zookeeper
  kafka-2:
    image: bitnami/kafka:2
    ports:
      - 9094:9094
    environment:
      KAFKA_CFG_ZOOKEEPER_CONNECT: zookeeper:2181
      ALLOW_PLAINTEXT_LISTENER: "yes"
      KAFKA_LISTENERS: >-
          INTERNAL://:29092,EXTERNAL://:9094
      KAFKA_ADVERTISED_LISTENERS: >-
          INTERNAL://kafka-2:29092,EXTERNAL://localhost:9094
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: >-
          INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: "INTERNAL"
    depends_on:
      - zookeeper
  kafdrop:
    image: obsidiandynamics/kafdrop:latest
    ports:
      - 9000:9000
    environment:
      KAFKA_BROKERCONNECT: >-
          kafka-0:29092,kafka-1:29092,kafka-2:29092
    depends_on:
      - kafka-0
      - kafka-1
      - kafka-2
```

Launch the stack with `docker-compose up`. This will take a few seconds to start, spinning up a single-node ZooKeeper ensemble with three Kafka brokers attached. The brokers will be externally bound to ports `9092`, `9093`, and `9094`; therefore, we must adjust our bootstrap list accordingly. Kafdrop is also bundled — exposed on port `9000`. Three brokers are a slight overkill for this example, but you may find it useful for other exercises.

First, we will create a test topic named `growth-plan`, with two partitions and a replication factor of one. Key in the command below.

```
$KAFKA_HOME/bin/kafka-topics.sh \
    --bootstrap-server localhost:9092,localhost:9093,localhost:9094 \
    --create --topic growth-plan --partitions 2 \
    --replication-factor 1
```

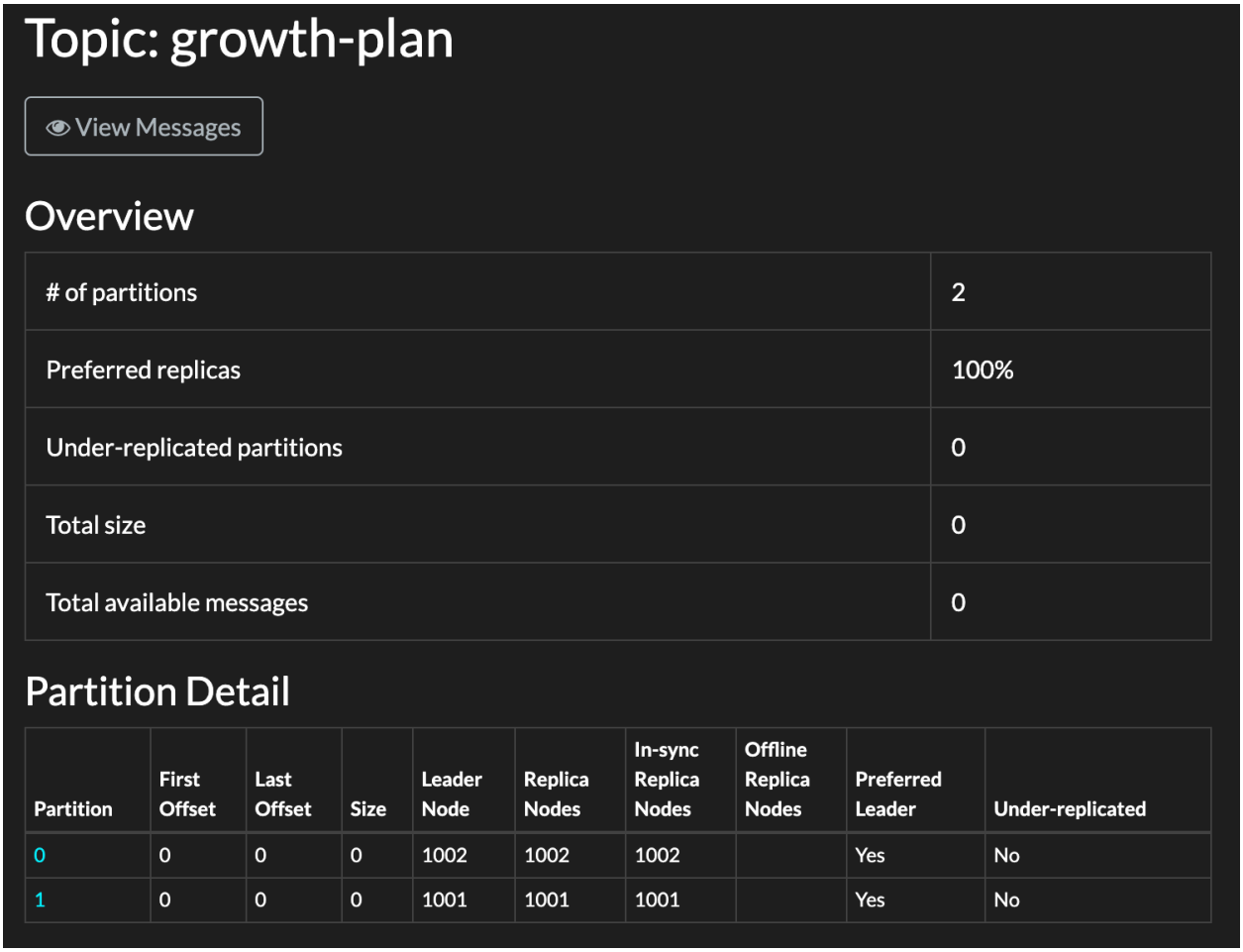
Having created the topic, examine it by running the `kafka-topics.sh` CLI tool:

```
$KAFKA_HOME/bin/kafka-topics.sh \
    --bootstrap-server localhost:9092,localhost:9093,localhost:9094 \
    --describe --topic growth-plan
```

The output indicates that there are two partitions, with their leadership assigned to two different brokers. As expected, the replication factor is one. (The assignment of replicas is random, so your output may vary from the one below.)

```
Topic: growth-plan  PartitionCount: 2 ReplicationFactor: 1 □
    Configs: segment.bytes=1073741824
  Topic: growth-plan  Partition: 0  Leader: 1002  Replicas: 1002 □
      Isr: 1002
  Topic: growth-plan  Partition: 1  Leader: 1001  Replicas: 1001 □
      Isr: 1001
```

Looking at Kafdrop, we can see a similar picture:



Kafdrop showing a topic with one replica

Let's increase the replication factor from one to two. We would like to keep the partitions balanced, letting the brokers alternate in the leader-follower status for each partition. Ideally, we don't want to disrupt any existing assignments, or change the leader status of any replicas. Broker `1002` should be the leader for partition `0`, and follower for partition `1`. Conversely, broker `1001` should lead partition `1` and follow partition `0`.

Having formed a plan in our head, it is time to capture it in a *reassignment file*. Create a file named `alter-replicas.json`, containing the following:

```
{
  "version": 1,
  "partitions": [
    {
      "topic": "growth-plan",
      "partition": 0,
      "replicas": [1002, 1001]
    },
    {
      "topic": "growth-plan",
      "partition": 1,
      "replicas": [1001, 1002]
    }
  ]
}
```

The reassignment file enumerates over all topic-partitions that require alteration, listing the new replicas as an array of numeric broker IDs. By convention, the first element in the array identifies the preferred leader.

Next, apply the reassignment file using the `kafka-reassign-partitions.sh` tool.

```
$KAFKA_HOME/bin/kafka-reassign-partitions.sh \
    --zookeeper localhost:2181 \
    --reassignment-json-file alter-replicas.json --execute
```

The `kafka-reassign-partitions.sh` tool is limited to working with ZooKeeper. It cannot be used without the `--zookeeper` flag.

The application of replica changes will echo the original assignments to the console as a JSON document. This can be saved as a JSON file in case it becomes necessary to revert to the original configuration.

```
Current partition replica assignment

{"version":1,"partitions":[{"topic":"growth-plan","partition":1, ⏋
"replicas":[1001],"log_dirs":["any"]},{"topic":"growth-plan", ⏋
"partition":0,"replicas":[1002],"log_dirs":["any"]}]}

Save this to use as the --reassignment-json-file option during ⏋
rollback
Successfully started reassignment of partitions.
```

Given the lack of any partition data in this simple example, the reassignment should be near-instant. This wouldn't necessarily be the case for a production topic, potentially containing lots of data; the reassignment time would depend on the size of the partitions, the available network bandwidth, and the performance of the brokers. To ascertain whether the replica changes completed successfully, run the `kafka-reassign-partitions.sh` tool with the `--verify` switch:

```
$KAFKA_HOME/bin/kafka-reassign-partitions.sh \
    --zookeeper localhost:2181 \
    --reassignment-json-file alter-replicas.json --verify
```

When all partitions have been successfully reassigned, the output should resemble the following:

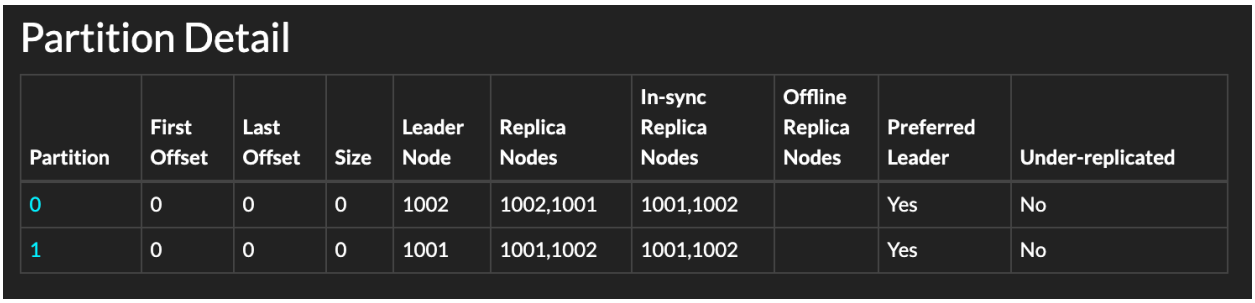
```
Status of partition reassignment:
Reassignment of partition growth-plan-0 completed successfully
Reassignment of partition growth-plan-1 completed successfully
```

Reassignment may involve copying large amounts of data over the network, which may affect the performance of the cluster. To throttle the replication process, run `kafka-reassign-partitions.sh` with the `-throttle` flag, specifying a cap on the desired bandwidth in bytes per second. The throttle will persist even after the initial replication completes, affecting regular replication for the topic in question. To lift the throttle, run the `kafka-reassign-partitions.sh` command with the `-verify` switch. The throttle will be cleared only if all partitions have had their reassignments completed successfully; otherwise, the throttle will remain in force.

Once reassignment completes, run the `kafka-topics.sh` command again. The output should now resemble the following.

```
Topic: growth-plan  PartitionCount: 2 ReplicationFactor: 2 □
    Configs: segment.bytes=1073741824
  Topic: growth-plan  Partition: 0  Leader: 1002 □
      Replicas: 1002,1001 Isr: 1002,1001
  Topic: growth-plan  Partition: 1  Leader: 1001 □
      Replicas: 1001,1002 Isr: 1001,1002
```

Similarly, Kafdrop will also reflect the new replica nodes.



Partition Detail

Partition	First Offset	Last Offset	Size	Leader Node	Replica Nodes	In-sync Replica Nodes	Offline Replica Nodes	Preferred Leader	Under-replicated
0	0	0	0	1002	1002,1001	1001,1002		Yes	No
1	0	0	0	1001	1001,1002	1001,1002		Yes	No

Kafdrop showing a topic with two replicas

Preparing reassignment files by hand can be laborious. There are several open-source helper scripts that automate this process. One such script is the `kafka-reassign-tool`, hosted on GitHub at github.com/dimas/kafka-reassign-tool[28].

Decommissioning broker nodes

The built-in `kafka-reassign-partitions.sh` tool might appear overly low-level and cumbersome for simple use cases such as adjusting the replication factor. However, there is a reason for that: this tool is designed to administer arbitrary changes to the replication topology. One such use case is for rebalancing partitions among broker nodes, levelling the load when new nodes are added to the cluster. Another use case is for moving partitions off a broker node before it can be safely decommissioned

Having selected a broker for decommissioning, create a reassignment file that covers all partitions for which the outgoing broker is a replica. Substitute the outgoing broker ID with one of the remaining brokers, ensuring that the original replication factor is preserved (unless you wish to change the replication factor in the process). Once preparations are complete, run `kafka-reassign-partitions.sh` with the `-execute` switch, then follow up with the `-verify` switch to ensure that the process completes. Having reassigned the partitions, run the `kafka-topics.sh` command with the `-describe` switch without specifying a topic name, to verify that the outgoing broker no longer features in any replicas, across any of the existing topics. Only after the broker has been completely fenced off, can it be permanently removed from the cluster.

[28] https://github.com/dimas/kafka-reassign-tool

Acknowledgements

The `acks` property stipulates the number of acknowledgements the producer requires the leader to have received before considering a request complete, and before acknowledging the write with the producer. This is fundamental to the durability of records; a misconfigured `acks` property may result in the loss of data while the producer naively assumes that a record has been stably persisted.

Although the property relates to the number of acknowledgements, it accepts an enumerated constant being one of —

- `0`: Don't require an acknowledgement from the leader.
- `1`: Require one acknowledgement from the leader, being the persistence of the record to its local log. This is the default setting when `enable.idempotence` is set to `false`.
- `-1` or `all`: Require the leader to receive acknowledgements from all in-sync replicas. This is the default setting when `enable.idempotence` is set to `true`.

Each of these modes is discussed in detail in the following subsections.

No acknowledgements

When `acks=0`, any records queued on the outgoing socket are assumed to have been published, with no explicit acknowledgements required from the leader. This option provides the weakest durability guarantee. Not only is it impacted by a trivial loss of a broker, but the availability of the client is also a determinant in the durability of the record. If the client were to fail, any queued records would be lost.

In addition to forfeiting any sort of durability guarantees, the client will not be informed of the offset of the published record. Normally, a client would obtain the offset in one of two different ways: supplying a `org.apache.kafka.clients.producer.Callback` to the `Producer.send()` method, or blocking on the result of a `Future<RecordMetadata>` object that is returned from `send()`. In either case, the resulting `RecordMetadata` would contain the offset of the published record. However, when `acks=0`, `RecordMetadata.hasOffset()` will return `false`, and `RecordMetadata.offset()` will return `-1`, signifying that no offset is available.

The main use case for `acks=0` is that the publisher can afford to lose either a one-off record or small contiguous batches of records with a negligible impact on the correctness of the overall system. Consider a remote temperature sensor that is periodically submitting telemetry readings to a server, which in turn, is publishing these to a Kafka topic. A downstream process will ingest these readings and populate a web-based dashboard in real-time.

In this hypothetical scenario, and provided that historical temperature readings are not significant, we can afford to lose some number of intermediate readings, provided that a reading for every sensor eventually comes through. This is not to say that a higher durability rating would somehow

be inapplicable, only that it may not be strictly necessary. Furthermore, by neglecting acknowledgements, we are also largely sidestepping the matters of error handling. This leads to a simpler, more maintainable application, albeit one that offers virtually no delivery guarantees.

Setting `acks=0` in no way implies that records will not be replicated by the leader to the in-sync replicas. Standard asynchronous replication behaviour applies identically in all cases, irrespective of the number of requested acknowledgements, provided that the record actually arrives at the broker. You can think of the `acks` property as the assurance level of a notional contract between three parties — the client, the partition leader, and the follower replicas. The contract is always present, with each party having the best intentions, but nonetheless, the contract may not be fulfilled if one of the parties were to fail. At its most relaxed level — `acks=0` — the extent of the assurance is constrained to the client. With each increase in the value of `acks`, the producer expands the scope of assurance to include the next party in the list. With `acks=all`, the producer will not be satisfied until all parties have fulfilled their contractual obligations.

One acknowledgement

When `acks=1`, the client will wait until the partition reader queues the write to its local log. The broker will asynchronously forward the record to all follower replicas and may respond to the publisher before it receives acknowledgements from all in-sync replicas.

On the face of it, this setting may appear to offer a significantly stronger durability guarantee over `acks=0`. If in the `acks=0` case the system was intolerant of the failure of a single party, in the `acks=1` case the failure of the partition leader should be tolerated, provided the client remains online. Should the leader fail, the producer can forward its request to another replica, whichever one takes the place of the leader.

In reality, this stance is ill-advised for several reasons. One is to do with how Kafka writes records to its log. Records are considered written when they are handed over to the underlying file system buffers. However, the broker does not invoke the `fsync` operation for each written record, in other words, it does not wait for the write to be flushed before replying to the client. Therefore, it is possible for the broker to acknowledge the write and fail immediately thereafter, losing the written chunk before it has been replicated onto the followers. The producer will continue on the assumption that the write has been acknowledged.

The second reason is a close relative of the first. Assume for the moment that the write was successfully committed to the leader's disk, but the leader suffered a failure before the record was received by any of the followers. The controller node, having identified the failure, will appoint a new leader from the remaining in-sync replicas. There is no guarantee that the new leader will have received the last few records that were queued for replication by the outgoing leader.

This is not to say that `acks=1` offers no additional durability benefits whatsoever, rather, the distinction between `acks=0` and `acks=1` is only discernible if there is a material difference between

the relative reliability of the client and broker nodes. And sometimes this is indeed the case: brokers may be assembled using higher-grade hardware components with more integrated redundancy, making them less failure-prone. Conversely, clients may be running on ephemeral instances that are susceptible to termination at short notice. But taking this for granted would be credulously presumptuous; it is prudent to treat a broker like any other component, assuming that if something can fail, it probably will, and when we least expect it to.

With the above in mind, set `acks=1` in those scenarios where the client can trivially afford to lose a published record, but still requires the offset of the *possibly* published record — either for its internal housekeeping or for responding to an upstream application.

All acknowledgements

When `acks=all` or `acks=-1` (the two are synonymous), the client will wait until the partition leader has gathered acknowledgements from the complete set of in-sync replicas. This is the highest guarantee on offer, ensuring that the record is durably persisted for as long as one in-sync replica remains. As it was stated earlier, the number of in-sync replicas may vary from the total number of replicas; the in-sync replicas are represented by a set that may grow and shrink dynamically depending on the observed replication lag between the leader and each follower. Since the latency of a publish operation when `acks=all` is attributable to the slowest replica in the ISR, the in-sync replica set is designed to minimise this latency by temporarily dismissing those replicas that are experiencing starvation due to I/O bottlenecks or other performance-impacting contingencies.

There is a pair of subtle configuration gotchas lurking in Kafka: one being on the client while the other on the broker. The default setting of `acks` is `1` when idempotence is disabled. (Idempotence is discussed in Chapter 10: Client Configuration.) This is unlikely to meet the durability expectations of most applications. While the rationale behind this decision is undocumented, the choice of `acks=1` may be related to the disclosure of `RecordMetadata` to the producer. With `acks=1`, the producer will be informed of the offset of the published record, whereas with `acks=0` the producer remains unaware of this. Kafka's own documentation states: "When used together, `min.insync.replicas` and `acks` allow you to enforce greater durability guarantees. A typical scenario would be to create a topic with a replication factor of `3`, set `min.insync.replicas` to `2`, and produce with `acks` of `all`." In spite of recognising `acks=all` as being necessary to fulfill a 'typical' scenario, the default value of `acks` remains `1`. The second 'gotcha': the default value of `min.insync.replicas` is `1`, again contradicting the 'typical' scenario.

Set `acks=all` in those scenarios where the correctness of the system is predicated on the consensus between the producer and the Kafka cluster as to whether a record has been published. In other words, if the client thinks that the record has been published, then it better be, or the integrity of the system may be compromised. This is often the case for transactional business applications where recorded events correlate to financial activity; the loss of a record may directly or indirectly incur financial losses.

The main drawback of `acks=all` is the increased latency as a result of several blocking network calls — between the producer and the leader, and between the leader and a subset of its followers. This is the price of durability.

Which setting is right for me?

The explanations above have clarified the behaviour of the producer client and the partition leader with respect to the value of the `acks` property. The reader should now have a better appreciation of the scenarios that may be fitting to the different acknowledgement modes.

If in doubt, set `acks` to `all`. This will ensure that a record is acknowledged by a number of brokers that is, *at minimum*, equal to the value of `min.insync.replicas`. Bear in mind that 'acknowledged' is not the same as 'stably persisted'. Failure of a broker immediately following an acknowledgement may result in the loss of data, therefore it is essential that the value of `min.insync.replicas` accurately reflects your tolerance for data loss. Setting `min.insync.replicas` to `2` on a topic with a replication factor of three, and using `acks=all` on the producer ensures that every published record is acknowledged by the leader and *at least* one other replica. The increased latency might be noticeable, but it may be preferable to data loss. One might even argue that if latency is the overarching concern that dominates technical decision-making, then Kafka is perhaps not the best solution for the problem at hand.

This chapter explored the innards of Kafka's internal replication protocol. We learned how Kafka utilises in-sync replicas to reduce the number of acknowledgements required for durable persistence, efficiently achieving distributed consensus.

We looked at how partition availability and data consistency concerns may be impacted by the replication setup, namely the inherent availability-consistency trade-offs present in deciding on the replication factor and the `min.insync.replicas` parameter. This ties into Kafka's leader election — the decisions made when promoting follower replicas in the event of leader failure.

Configuration of the replication topology was then explored. We worked through examples of using the built-in `kafka-reassign-partitions.sh` tool to affect fine-grained control over a partition's replicas, and its potential use cases outside of simple replication factor adjustments. The performance trade-offs of durable persistence were covered, and we delved into the idiosyncrasies of Kafka's default configuration settings, and how they may catch out unsuspecting users.

Finally, durability was explored from the eyes of the producer. We looked at how the `acks` producer setting impacts the durability guarantees of individual records, and its interplay with the replication factor and the `min.insync.replicas` broker-side settings.

Chapter 14: Data Retention

Given that infinite storage is yet be invented, what happens to all those records stored in Kafka?

Chapter 3: Architecture and Core Concepts introduced the concept of low and high-water marks and how these shift as records are written to the log and as records are eventually purged. The focus on this chapter will be the latter; specifically, the conditions under which data is removed and the precise behaviour of Kafka in that regard.

Kafka storage internals

Given the option, most operators prefer to treat Kafka as a black box when configuring its data retention behaviour, and many other aspects, for that matter. By all means, this is understandable — being able to quickly consult the official documentation for a handful of configuration properties, apply them as stated in the instructions and move on to a less mundane task — almost sounds too reasonable to be true. And unfortunately, it is; Kafka is brimming with nuanced behaviour that is strongly coupled to the underlying implementation. Without the necessary insights, one may easily lose many days trying to explain the reasons that Kafka's behaves the way it does, seemingly in contradiction to the way it was configured.

Organisation of log data

Before exploring how Kafka's data retention mechanisms operate, it is instructive to gain a basic understanding of its log structure. As it has been stated numerous times, Kafka is a distributed, append-only log. The *distributed* aspect was covered in Chapter 13: Replication and Acknowledgements. Turning to the local persistence of a partition — which, as we know, occurs at each replica — the log is organised on disk as a series of data and index files.

The log files for each replicated partition are stored in a dedicated subdirectory of `log.dirs`, which points to `/tmp/kafka-logs` by default. The subdirectory is named by concatenating the topic name with the partition index, delimited by the `-` (hyphen) character. For example, partition `0` of the `getting-started` topic can be found at `/tmp/kafka-logs/getting-started-0`. Listing the contents of the directory with the `tree` command, we can see the following:

```
.
├── 00000000000000000000.index
├── 00000000000000000000.log
├── 00000000000000000000.timeindex
└── leader-epoch-checkpoint
```

The `leader-epoch-checkpoint` file contains the leader epoch — the number of times the leadership of a partition has been reassigned by the cluster controller. The replicas use the leader epoch as a means of verifying the current leader, fencing a prior leader if their leadership status has been superseded. If a displaced leader attempts to syndicate data, the replicas will sense that it is operating under an outdated epoch number and will ignore its commands.

Indexes

The files named with a large, zero-padded integer are *log segments*. Kafka does not store its partition log in a single, contiguous 'jumbo' file. Instead, logs are broken down into discrete chunks. Each segment is named in accordance with the offset of the first records that it contains. This is called the *base offset*.

Records may be of arbitrary size, and therefore occupy a variable amount of space in their respective `.log` file. To further complicate matters, records are persisted in batches (which might be compressed). Variable record size and batching mean that a record cannot be trivially located by its offset using the `.log` file alone. This is where the `.index` file comes in. Essentially, an index is a sorted map of record offsets to the physical locations of the corresponding records in the `.log` file. Where a record is encompassed within a batch, the index file points to the beginning of the batch entry. For compactness, the index file uses a 32-bit integer to store offsets, having subtracted the base offset from the record's offset beforehand. The physical locations pointed to by the index are also stored as 32-bit integers, for a combined entry size of 8 bytes. Indexes are memory-mapped files; the storage space for an index is preallocated when the index file is created. The amount of preallocated space is prescribed by the `log.index.size.max.bytes` property, which defaults to `10485760` (10 MiB) and acts as a default value for all topics. To control the extent of the preallocation for a specific topic, set the `segment.index.bytes` property. When a log file is rolled over, the remaining unused space in its index files is returned to the filesystem.

Kafka uses different names for properties that affect the default behaviour versus those that target individual topics. There is no particular naming convention to these properties; for example, the `log.roll.ms` cluster-wide property corresponds to the `segment.ms` per-topic property.

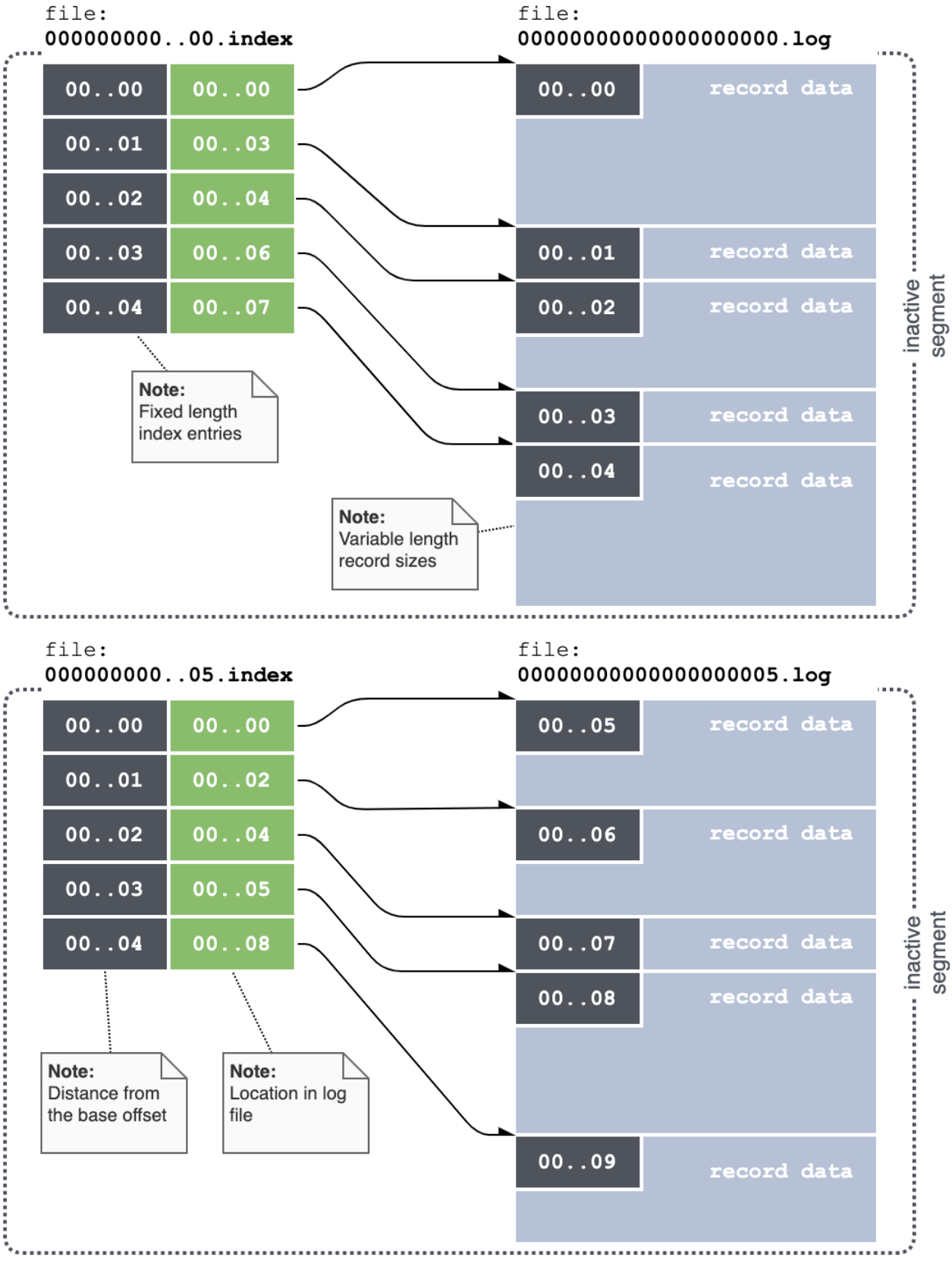
Not every record in the log has a corresponding entry in the index. For starters, records may be part of a batch, and only the location of the batch is indexed. In addition, to constrain the size of the index files and to minimise the amount of additional disk I/O, records are indexed at an interval

specified by the `log.index.interval.bytes`, which is `4096` by default. To override this setting on a per-topic basis, the `index.interval.bytes` property can be assigned on the topic entity.

With the aid of an index file, it is possible to locate a record in *O(1)* complexity. Given a record's offset — by subtracting the base offset, dividing by the index interval, and seeking to the resulting location in the index file, the physical location of the record is revealed. Where the record in question might not be indexed, the previously indexed offset is employed, requiring a short scan to find the record.

The `.timeindex` file is a close cousin of the `.index` file, mapping the millisecond-precise UNIX epoch timestamp of each record to its location in the `.log` file. The presence of `.timeindex` files allows Kafka to locate records based on time, rather than an offset.

The diagram below illustrates the segmented log structure and the use of indexes to map record offsets to physical locations in the log files.


file:
000000000..00.index
00..00
00..00
00..01
00..03
00..02
00..04
00..03
00..06
00..04
00..07
Note:
Fixed length
index entries
file:
00000000000000000000.log
00..00
record data
00..01
record data
00..02
record data
00..03
record data
00..04
record data
Note:
Variable length
record sizes
inactive
segment
file:
000000000..05.index
00..00
00..00
00..01
00..02
00..02
00..04
00..03
00..05
00..04
00..08
Note:
Distance from
the base offset
Note:
Location in log
file
file:
00000000000000000005.log
00..05
record data
00..06
record data
00..07
record data
00..08
record data
00..09
record data
inactive
segment
file:
000000000..10.index
00..00
00..00
00..01
00..01
file:
00000000000000000010.log
00..10
record data
00..11
record data
active
segment

Rotation of log segments

With the basics covered, we arrive at the most crucial part of our preamble — the rotation of log segments. Specifically, it is the log rotation behaviour that has the greatest potential to cause confusion in relation to Kafka's data retention mechanisms that will be discussed shortly.

A replica maintains a single *active* log segment for every partition that it hosts. The active segment corresponds to the *head end* of the log. All writes performed on behalf of producers are appended to the active log segment, while consumer reads may occur from any segment in the log. For a given active segment, no entity other than a producer is permitted to modify the segment and its corresponding index files; this constraint is positively intuitive in relation to the append-only nature of a log in Kafka.

At some point, the active segment file will approach a set of predefined constraints that, when breached, will trigger a roll-over. These constraints are defined by the following broker properties:

- `log.segment.bytes` — the maximum size of a single log file. Defaults to `1073741824` (1 GiB).
- `log.roll.hours` — the maximum amount of time (in hours) that an active segment is allowed to exist before being closed. The default value is `168` (one week). The age of a segment is derived by subtracting the timestamp of the first record from the current time. The size-based and time-based properties are complementary — tripping either of these thresholds is sufficient to initiate a roll-over.
- `log.roll.ms` — where the use of hour multiples to specify the roll-over threshold does not provide sufficient granularity, this property may be used to specify the threshold with millisecond precision. When set, this property supersedes `log.roll.hours`.

Replicas typically host multiple log files, particularly when dealing with 'wide' topics (i.e. topics with lots of partitions). It is likely that when a log segment reaches its roll-over threshold, other log segments would have reached theirs, or are just about to. To prevent a stampede of I/O requests, a broker will introduce a random amount of *jitter* — an artificial delay from the time a roll-over was triggered to the time of enacting the roll-over — thereby spreading the correlated roll-over of multiple log segments over a larger time period. The jitter values apply to the `log.roll.hours` and `log.roll.ms` settings, and are named `log.roll.jitter.hours` and `log.roll.jitter.ms`, respectively. No jitter is applied by default.

The properties listed above represent the baseline configuration applicable to all topics, and may be updated statically or dynamically, but cannot be used to target individual topics. There are equivalent per-topic properties that can be set dynamically:

- `segment.bytes` — in place of `log.segment.bytes`.
- `segment.ms` — in place of `log.roll.ms`.
- `segment.jitter.ms` — in place of `log.roll.jitter.ms`.

While index files are preallocated by default, log segments are not — owing to the log segment being rather large (1 GiB by default). Also, Linux-based filesystems are generally performant with respect to append operations. The lack of preallocation has been known to cause performance issues when Kafka is run on NTFS and some older Linux filesystems. To enable preallocation of log files, set the `log.preallocate` property to `true` for all topics, or the `preallocate` property when targetting individual topics.

Viewing log contents

Both log files and index files are encoded in binary form for compactness. Reading the data directly will not translate to anything particularly meaningful to the user, with the possible exception of the key or value portions of a record if these happen to be strings. To assist in diagnosing issues with logs and indexes, Kafka provides a CLI tool — `kafka-dump-log.sh`. This utility must be run locally on the broker housing the partition data, and can operate on either file type (including time indexes). The example below shows the output of `kafka-dump-log.sh` over a segment of the `getting-started-0` log.

```
LOG_DIR=/tmp/kafka-logs &&
$KAFKA_HOME/bin/kafka-dump-log.sh \
    --files $LOG_DIR/getting-started-0/00000000000000000000.log \
    --print-data-log
```

```
Dumping /tmp/kafka-logs/getting-started-0/00000000000000000000.log
Starting offset: 0
baseOffset: 0 lastOffset: 0 count: 1 baseSequence: -1 □
    lastSequence: -1 producerId: -1 producerEpoch: -1 □
    partitionLeaderEpoch: 0 isTransactional: false □
    isControl: false position: 0 CreateTime: 1577930118584 □
    size: 73 magic: 2 compresscodec: NONE crc: 2333002560 □
    isvalid: true
| offset: 0 CreateTime: 1577930118584 keysize: -1 valuesize: 5 □
    sequence: -1 headerKeys: [] payload: alpha
```

Deletion

Reverting to the original question, what happens when we accumulate too much data?

Kafka offers two independent but related strategies — called *cleanup policies* — for dealing with data retention. The first and most basic strategy is to simply *delete* old data. The second strategy is more elaborate — *compact* prior data in such a manner as to preserve the most amount of information.

Both strategies can co-exist. This section will focus on the delete strategy; the following section will address compaction.

The cleanup policy can be assigned via the `log.cleanup.policy` broker property, which takes a comma-separated list of policy values `delete` and `compact`. The default is `delete`. This setting can be assigned statically or dynamically, acting as a cross-topic default. A topic-specific policy can be assigned via the `cleanup.policy` property — overriding the default `log.cleanup.policy` for the topic in question.

The `delete` policy operates at the granularity of log segments. A background process, operating within a replica, looks at each inactive log segment to determine whether a segment is eligible for deletion. The thresholds constraining the retention of log segments are:

- `log.retention.bytes` — the maximum allowable size of the log before its segments may be pruned, starting with the tail segment. This value is not set by default as most retention requirements are specified in terms of time, rather than size.
- `log.retention.hours` — the number of hours to keep a log segment before deleting it. The default value is `168` (one week). The size-based and time-based properties are complementary; the breach of any of the two constraints is sufficient to trigger segment deletion.
- `log.retention.minutes` — specifies the retention in minutes, where hour multiples fail to provide sufficient granularity. If set, this property overrides the `log.retention.hours` property.
- `log.retention.ms` — specifies the retention in milliseconds, overriding the `log.retention.minutes` property.

The background process checks files at an interval specified by `log.retention.check.interval.ms`, defaulting to `300000` (five minutes). This is sufficient for most retention settings where the lifetime of a log is measured in hours, days or weeks. When testing the deletion policy over smaller time frames, it may be necessary to wind this setting down to decrease the time between successive checks.

As it was stated in Chapter 3: Architecture and Core Concepts, the deletion of records results in the advancement of the *beginning offsets* (also known as the low-water mark) to the point of the first preserved record. This happens automatically as log segments are deleted.

Because the granularity of the `delete` policy is a log segment, it may be necessary to adjust the maximum size of a segment depending on one's expectations around the responsiveness of the deletion process. Suppose there was a requirement that a record should not be kept longer than seven days, but at least five days of retention is required. Setting `log.retention.hours` to `168` (seven days) would satisfy the lower bound of five days, but depending on how long it takes for the active log segment to roll over, the upper bound might not be satisfied. Recall, only producer-initiated writes are allowed on the active segment; all other manipulations — deletion and compaction — may only occur on the inactive segments. With the default settings in place, log files are not rotated until they either reach a maturity of seven days or grow to over 1 GiB in size. To satisfy the upper bound, we must constrain the lifetime of a log segment, so that it forcibly rolls over within a more suitable time frame. For example, we might set `log.retention.hours` to `132` (five days and twelve hours) and `log.roll.hours` to `24`, thereby forcing a collection after six-and-a-half days.

The `log.roll.hours` setting (and its more fine-grained equivalents) are an effective measure for the predictable rotation of log files in low-throughput topics, where there is insufficient write pressure to cause a roll-over due to file size alone. However, even the time-based trigger requires at least one write to the active segment for the broker to realise that its time is up, so to speak. If a topic is experiencing no writes for whatever reason, Kafka will not proactively rotate the active log segment, even if the latter has outlived its expected lifespan. So in the example above, we can only satisfy the upper bound requirement if there is a constant trickle of records to the topic in question.

Log deletion applies not just to user-defined topics, but also internal topics such as `__consumer_offsets`. For more information on how the deletion of log segments may affect consumer groups, see Chapter 5: Getting Started.

Compaction

Use cases behind compaction

Overwhelmingly, Kafka is used as a replicated log for notifying downstream parties of events of certain relevance, where the latter typically correlate to changes in an upstream system of record. Often, this system is backed by a database or some other persistent data store that offers an interface for querying the current state of the data, should the need for a holistic view of the state snapshot arise. Consumers may build their bespoke projections of the upstream state by faithfully replicating all relevant changes, provided these are emitted as events. At either rate, Kafka is used as an intermediate transport of sorts — essential middleware that underpins a broader, event-driven system. And as essential as it may appear in this context, in the majority of its uses, Kafka does not entirely replace a system of record.

Although as one might argue, Kafka comes remarkably close to fulfilling the role of a data store — particularly as applications grow more dependent on a replicated log. In some event-driven systems, the notion of querying the master data store has been virtually eliminated or, more often, relegated to the *hydration* scenario, where first-time consumers may query the master data in order to prime their internal projection, and switch to Kafka-based replication thereafter. For the reader's reference, such systems are called *bimodal* — in that they employ one mode for the initial *hydration* and another for the subsequent *replication*. While they are mostly straightforward — in that both modes can be easily reasoned about — there is the added complexity of maintaining code that deals with a one-off event — a contingency that typically occurs once in the lifetime of an application in its target operating environment.

What if Kafka was going to be used for event sourcing? Specifically, in the role of a definitive event store that any consumer could use to reconstitute the state of an entire domain from first principles — by replaying all updates from the point of conception. Such an arrangement would have a clear benefit — *unimodality*; specifically, using one mode of operation to accommodate both the *hydration* scenario and the subsequent *replication* of event data. Simpler code; easier to test; easier to maintain.

An event store recording all changes to every entity of interest over its entire lifetime? That sounds like a scalability nightmare. Where would one find the space to store all this data? And how long would the initial hydration take — having to painstakingly replay all records from day dot?

Even with infinite retention, the naive approach of storing everything would hardly be practical — especially for fast-moving data where any given entity might be characterised by a considerable number of updates over its lifetime. The replay time would grow beyond comprehension; we would have ourselves an event store that no one would dare read from. On the flip side, discarding the oldest records, while binding the replay time, would result in the loss of data — forcing us back down the bimodal path.

Kafka accommodates the classical unimodal event sourcing scenario using its log compaction feature. Where deletion operates at the segment level, compaction provides a finer-grained, per-record culling strategy — as opposed to the coarser-grained time-based retention. The idea is to selectively prune a related set of records, where the latter represents updates to some long-lived entity, and where a more recent (and therefore, more relevant) update record exists for the same key. In doing so, the log is guaranteed to have at least one record that represents the most recent snapshot of an entity — sacrificing the timeline of updates in order to maximally preserve the current state.

Another way of looking at compaction is that it turns a stream into a snapshot, where only the most recent information is retained. This is akin to a row in a database, where the latter disregards the timeline of updates — keeping only the last value. For this to work, the granularity of an update must align with the granularity of the underlying entity; in other words, *updates must be fully self-contained*. Compaction is intractable when updates contain *deltas* — piecemeal information that can be used to incrementally build the state of an entity. If a more recent update supersedes its logical predecessor, *there cannot be any information contained within the preceding record that cannot be obtained by inspecting the successor.*

Overwriting and deleting records

Compaction is activated by adding `compaction` to the list of policies in `log.cleanup.policy` (or `cleanup.policy` for the per-topic setting). Apart from adding the policy, compaction requires the *cleaner* process, which is enabled by default and controlled by the `log.cleaner.enable` property.

Once compaction is activated, there is nothing more required from the producer to overwrite an older record – one simply publishes an update with the same key. With compaction in force, the number of records approaches the number of entities that are being described. When the useful lifespan of an entity nears its conclusion, any previous updates for that entity may be deleted by publishing a *tombstone* — a record for the same key as the prior updates but with a `null` value. Tombstones will trigger the purging of like records, but will themselves be retained for some time before eventually being purged. (The reason for this will be discussed shortly.) There is nothing special about tombstones from a consumer's perspective — it will see records with `null` values — exactly as they were written by the producer.

Once a topic has been fully compacted (old records removed and tombstones purged), the amount of time a consumer will take to traverse the topic end-to-end will be proportional to the number of

retained unique keys. This is precisely what is needed to sustainably support the majority of event sourcing scenarios.

Behind the scenes

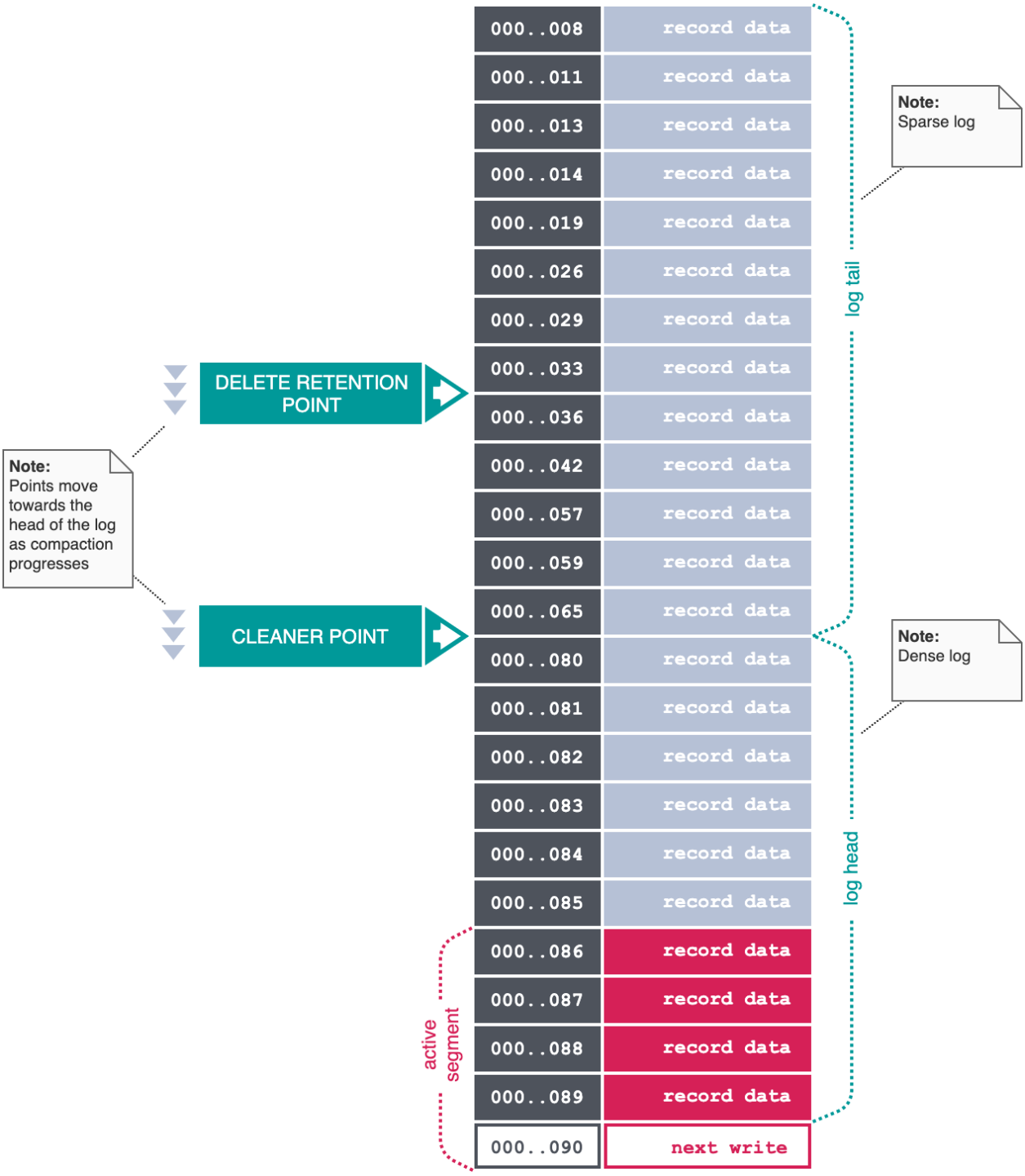
Compaction is fundamentally asynchronous, operating in the background by trailing the inactive log segments. Under the hood, log compaction recruits dedicated threads on each replica — the exact number of threads is given by the `log.cleaner.threads` broker property (defaulting to `1`). Compaction is driven by four main properties:

- `log.cleaner.min.cleanable.ratio` — the minimum ratio of the *dirty* log size to the total log size for a log to be eligible for cleaning. The default value is `0.5`. More on the dirty ratio in a moment.
- `log.cleaner.min.compaction.lag.ms` — prevents records newer than a minimum age from being subjected to compaction. The default value is `0`, implying there is no minimum time that a record will persist before it may be compacted, subject to the constraint of the active log segment. (Recall, only records in inactive segments are subject to manipulation.)
- `log.cleaner.max.compaction.lag.ms` — the upper bound on the compaction lag. Typically used with low-throughput topics, where logs might not exceed the `log.cleaner.min.cleanable.ratio` and, as such, will remain ineligible for compaction for an unbounded duration. The default setting is `Long.MAX_VALUE`, implying that compaction is driven by the `log.cleaner.min.cleanable.ratio`. (Note, the upper bound is not a hard deadline, being subject to the availability of cleaner threads and the actual compaction time.)
- `log.cleaner.delete.retention.ms` — the duration that the tombstone records are persisted before being purged from the log. More on tombstone retention in a moment.

The settings above act as cluster-wide defaults. Their per-topic equivalents are:

- `min.cleanable.dirty.ratio` — in place of `log.cleaner.min.cleanable.ratio`.
- `min.compaction.lag.ms` — in place of `log.cleaner.min.compaction.lag.ms`.
- `max.compaction.lag.ms` — in place of `log.cleaner.max.compaction.lag.ms`.
- `delete.retention.ms` — in place of `log.cleaner.delete.retention.ms`.

For every log being compacted, the cleaner maintains two logical points — the *cleaner point* and the *delete retention point*. The cleaner point is the location in the log corresponding to the progress of the cleaner through the log, dividing the log into two parts. The part older than the cleaner point is called the *log tail* and corresponds to those segments that have already been compacted by the cleaner. Conversely, the part of the log that is newer than the cleaner point is called the *log head* and corresponds to those segments that have yet to be compacted, including the active segment. The diagram below illustrates the location of these points in relation to the overall log.



Log compaction

The cleaner thread prioritises the log with the highest ratio of the head size to the overall length of the log — known as the *dirty ratio*. For a log to be selected for compaction, its dirty ratio must exceed the `log.cleaner.min.cleanable.ratio` or, alternatively the oldest uncompacted record must exceed the `log.cleaner.max.compaction.lag.ms`.

To perform the compaction, the cleaner scans the segments in the head portion, including the active segment, populating an in-memory hash table of unique record keys mapped to their most recent offsets in the log. All inactive log segments and their indexes are recopied (including those that have been previously compacted), purging records that have been superseded – except for tombstones, which are treated differently to conventional records. As the log cleaner works through the log segments, new segments get swapped into the log, replacing older segments. This way compaction does not require double the space of the entire partition; only the free disk space for one additional log segment is required.

Compaction preserves the order of the surviving records and their original offset; it will never reorder records or manipulate them in a way that might confuse a prospective consumer or contradict the producer's original intent. In effect, compaction converts a densely populated log into a sparse one, with 'holes' in offsets that should be disregarded by consumers. Reverting to what was stated in Chapter 3: Architecture and Core Concepts, a consumer should not interpret the offsets literally — using offsets purely as a means of inferring relative order.

The delete retention point lags behind the cleaner, prolonging the lifetime of tombstone records by an additional `log.cleaner.delete.retention.ms` on top of the `log.cleaner.min.compaction.lag.ms` accorded to conventional records. The `log.cleaner.delete.retention.ms` property applies equally to all topics, defaulting to `86400000` (24 hours); its per-topic equivalent is `delete.retention.ms`.

Tombstone records are preserved longer by design. The motivation for preserving tombstones is revealed in the following scenario. A publisher emits records for an entity, characterised by `Create` and `Update` events. Eventually, when the entity is deleted, the producer will follow with a tombstone — indicating to the compactor and the consumer ecosystem that the entity in question should be completely purged from the topic and any projections. This makes sense for a consumer that has not seen any traces of the entity. If the entity has already been dropped, then processing `Create`, `Update` and tombstone records for it is a waste of the consumer's time. It is more than a trivial waste of resources; in fact, it runs contrary to the requirements of a pure event store — the consumer should only be exposed to events that are legitimately required to reconstitute the snapshot of the current state, having been initialised with a clean slate.

The issue with deleting tombstones becomes apparent when considering a consumer operating with minimal lag that is part-way through handling entity updates for a soon-to-be-deleted entity. Suppose the producer has published `Create` and `Update` events for some entity, which the consumer has caught up with. The consumer then begins to accrue lag (perhaps it has gone offline momentarily). Meanwhile, the producer writes a tombstone record for the entity in question. If the compactor kicks in before the consumer resumes processing, the removal of all records for the key referenced by the tombstone would create an inconsistency on the consumer — leaving it in a state where the entity is retained in perpetuity. The handling of tombstones creates a contradiction between the requirements of hydration and subsequent replication — the former requires the tombstones to be deleted, while the latter needs tombstones in place to maintain consistency.

By allowing an additional grace period for tombstone records, Kafka supports both use cases. The tombstone retention time should be chosen such that it covers the maximum lag that a consumer might reasonably accumulate. The default setting of 24 hours may not be sufficient where consumers

are intentionally taken offline for long periods; for example, non-production environments. Also, it might not be sufficient for systems that need to tolerate longer periods of downtime; for example, to account for operational contingencies.

Compaction is a resource-intensive operation, requiring a full scan of the log each time it is run. The purpose of the `log.cleaner.min.cleanable.ratio` property (and its per-topic equivalent) is to throttle compaction, preventing it from running continuously, in detriment to the broker's normal operation. In addition to the cleanable ratio, the compactor can further be throttled by applying the `log.cleaner.backoff.ms` property, controlling the amount of time the cleaner thread will sleep in between compaction runs, defaulting to `15000` (15 seconds). On top of this, the `log.cleaner.io.max.bytes.per.second` property can be used to apply an I/O quota to log cleaner thread, such that the sum of its read and write disk bandwidth does not exceed the stated value. The default setting is `Double.MAX_VALUE`, which effectively disables the quota.

An example

To demonstrate the Kafka compactor in action, we are going to create a topic named `prices` with aggressive compaction settings:

```
$KAFKA_HOME/bin/kafka-topics.sh --bootstrap-server localhost:9092 \
    --create --topic prices --replication-factor 1 --partitions 1 \
    --config "cleanup.policy=compact" \
    --config "delete.retention.ms=100"  \
    --config "segment.ms=1" \
    --config "min.cleanable.dirty.ratio=0.01"
```

Normally, setting such small segment sizes and aggressive cleanable ratios is strongly discouraged, as it creates an obscenely large number of files and leads to heavy resource utilisation — every new record will effectively mark the partition as dirty, instigating compaction. This is done purely for show, as with conservative settings the compactor will simply not kick in for our trivial example. (The records will be held in the active segment, where they will remain out of the compactor's reach.)

Next, run a console producer:

```
$KAFKA_HOME/bin/kafka-console-producer.sh \
    --broker-list localhost:9092 --topic prices \
    --property parse.key=true --property key.separator=:
```

Copy the following line by line, allowing for a second or two in between each successive write for the log files to rotate. Don't copy-paste the entire block, as this will likely group the records into a batch on the producer, placing them all into the active segment.

```
AAPL:279.74
AAPL:280.03
MSFT:157.14
MSFT:156.01
AAPL:284.90
IBM:100.50
```

Press `CTRL-D` when done.

Now run the consumer. Assuming that the compactor has had a chance to run, the output should be constrained to the unique record keys and their most recent values.

```
$KAFKA_HOME/bin/kafka-console-consumer.sh \
    --bootstrap-server localhost:9092 \
    --topic prices --from-beginning \
    --property print.key=true
```

```
MSFT  156.01
AAPL  284.90
IBM   100.50
```

Also, we can see the effects of compaction on the log files:

```
tree -s /tmp/kafka-logs/prices-0
```

```
/tmp/kafka-logs/prices-0
├── [          0]  00000000000000000000.index
├── [        156]  00000000000000000000.log
├── [         12]  00000000000000000000.timeindex
├── [         10]  00000000000000000001.snapshot
├── [         10]  00000000000000000002.snapshot
├── [         10]  00000000000000000003.snapshot
├── [         10]  00000000000000000004.snapshot
├── [   10485760]  00000000000000000005.index
├── [         77]  00000000000000000005.log
├── [         10]  00000000000000000005.snapshot
├── [   10485756]  00000000000000000005.timeindex
└── [          8]  leader-epoch-checkpoint

0 directories, 12 files
```

The active log segment is `00000000000000000005.log`. All prior segments have been compacted, leaving hollow snapshots in their place; the remaining unique records were coalesced into `00000000000000000000.log`. Out of interest, we can view its contents using the `kafka-dump-logs.sh` utility:

```
$KAFKA_HOME/bin/kafka-dump-log.sh --print-data-log \
    --files /tmp/kafka-logs/prices-0/00000000000000000000.log


Dumping /tmp/kafka-logs/prices-0/00000000000000000000.log
Starting offset: 0
baseOffset: 3 lastOffset: 3 count: 1 baseSequence: -1 □
    lastSequence: -1 producerId: -1 producerEpoch: -1 □
    partitionLeaderEpoch: 0 isTransactional: false □
    isControl: false position: 0 CreateTime: 1577409425248 □
    size: 78 magic: 2 compresscodec: NONE crc: 2399134455 □
    isvalid: true
| offset: 3 CreateTime: 1577409425248 keysize: 4 valuesize: 6 □
    sequence: -1 headerKeys: [] key: MSFT payload: 156.01
baseOffset: 4 lastOffset: 4 count: 1 baseSequence: -1 □
    lastSequence: -1 producerId: -1 producerEpoch: -1 □
    partitionLeaderEpoch: 0 isTransactional: false □
    isControl: false position: 78 CreateTime: 1577409434843 □
    size: 78 magic: 2 compresscodec: NONE crc: 2711186080 □
    isvalid: true
| offset: 4 CreateTime: 1577409434843 keysize: 4 valuesize: 6 □
    sequence: -1 headerKeys: [] key: AAPL payload: 284.90
```

This is showing us that the `MSFT` and `AAPL` records have been bunched up together, although originally they would have appeared in different segments. The `IBM` record, having been published last, will be in the active segment.

The existence of the active segment is probably the most confusing aspect with respect to compaction, catching out its fair share of practitioners. When a topic appears to be partially compacted, in most cases the reason is that the uncompacted records lie in the active segment, and are therefore out of scope.

Combining compaction with deletion

As it was previously stated, it is possible to assign both `delete` and `compact` policies to the `log.cleanup.policy` property (or `cleanup.policy`, as the case may be). The question is: does it make sense to do so?

When a topic is configured for bounded retention, the typical use cases that are being serviced are bimodal replication of events or conventional event stream processing — where a consumer ecosystem or a series of processing stages react to events from an upstream emitter. Unbounded retention is typically a surplus to requirements; it is sufficient for the retention time to cover the maximum forecast consumer lag.

Conversely, when a topic is compacted, one naturally assumes that it is being utilised as the source of truth — where the preservation of state is an essential attribute — more so than the preservation of discrete changes. Under this model, there is either a characteristic absence of a queryable data source, or it may be intractable for the consumer to issue queries. This model enables unimodal processing, which is its dominant advantage.

Combining the two strategies might appear counterintuitive at first. It has been said: a compacted topic can be logically equated to a database (or a K-V store) — the most recent updates corresponding to rows in a table of finite size. Slapping a deletion policy on top feels like we are building a database that silently drops records when it gets to a certain size (or age). This configuration is peculiar, at the very least.

Nonetheless, a size-bound, compacted topic is useful in limited cases of change data capture and processing. This can be seen as an extension of the conventional, event-driven replication model. In the presence of fast-moving data, where updates are self-contained and the time-value of data is low, the use of compaction can dramatically accelerate the processing of data. More often than not, entity deletion is a non-issue, and tombstones are excluded from the model (although they don't have to be). Here, the intention is not to conserve space or to enable unimodal processing; rather, it is to reduce resource wastage on processing events that rapidly lose their significance. Because the size of the topic is inherently constrained (by way of deletion), one can afford to run a more aggressive compactor, using a much lower `log.cleaner.min.cleanable.ratio` in conjunction with smaller segment sizes. The more aggressive the compaction, the lower the likelihood that an obsolete record is observed by a consumer.

An example of where this 'hybrid' model is used is the internal `__consumer_offsets` topic, used to manage state information for consumer groups. This topic's segments are subjected to both compaction and deletion. Compaction allows for the rapid reconstruction of consumer state on the group coordinator, while deletion binds the size of the topic and acts as a natural 'garbage collector' of consumer groups that have not been utilised for some time.

This chapter looked at how Kafka deals with data retention. We started with a tour of its storage internals — understanding how topic-partition data on the replicas maps to the underlying filesystem. Among the key takeaways is the notion of a segmented log, whereby the partition is broken up into a series of chunks — each accompanied by a set of indexes — permitting *O(1)* lookups of records by their offset or timestamp. Of the log segments, the most recent is the *active* segment. Only a producer may write to the active segment — a fundamental design decision that influences the remainder of Kafka's data retention machinery. And also, one that frequently causes confusion.

Kafka offers two independent but related cleanup policies — *deletion* and *compaction*. Deletion is relatively straightforward — truncating log segments when the log reaches a certain size or where the oldest record matures to a certain age. As a cleanup policy, deletion is suitable where consumers

are expected to keep up with the data syndicated over a topic, or where an alternate mechanism for retrieving any lapsed data exists.

Compaction is the more elaborate cleanup policy, providing a fine-grained, per-record retention strategy. The idea is to selectively remove records where the latter represent entity updates, and where a more recent update record exists for the same key. Compaction leaves at least one record that represents the most recent snapshot of an entity — turning an event stream into a continuously updating data store.

Crucially, compaction is a *best-effort* endeavour — it ensures that at least one record for a given key is preserved. There is no *exactly-one* guarantee when it comes to compaction. Depending on various factors — the contents of the active segment and the dirty ratio — compaction might not run, or having run, it might appear to have a partial effect. Also, compaction is not bound by a deadline — being subject to the availability of cleaner threads, the backlog of competing partitions in need of compaction, and the actual compaction time.

The deletion and compaction policies may be used in concert, which is typically done to expedite the processing of change records while constraining the size of the log.

Among them, Kafka's cleanup policies provide sufficient flexibility to accommodate a range of data retention needs — from event-driven services to accelerated stream processing graphs and fully-fledged event-sourcing models.

Chapter 15: Group Membership and Partition Assignment

The reader should be aware by now that partitions in a topic are allocated approximately evenly among the live members of a consumer group, and that partition allocations for one group are entirely independent of the allocations of any other group. On the whole, group membership is pivotal to Kafka's architecture; it acts as a load balancing mechanism and ensures that any given partition is assigned to at most one member of a group.

This chapter explores Kafka's group membership protocol and the underlying mechanisms by which Kafka ensures that members are always able to make progress in the consumption of records, and that records are processed in the prescribed order and on at most one consumer in a group.

Parts of this chapter contain sizable doses of theory on Kafka's inner workings, and might not immediately feel useful to a Kafka practitioner. Nonetheless, an appreciation of the underlying mechanics will go a long way in understanding the more practical material, such as session timeouts, liveness, partition assignment, and static membership.

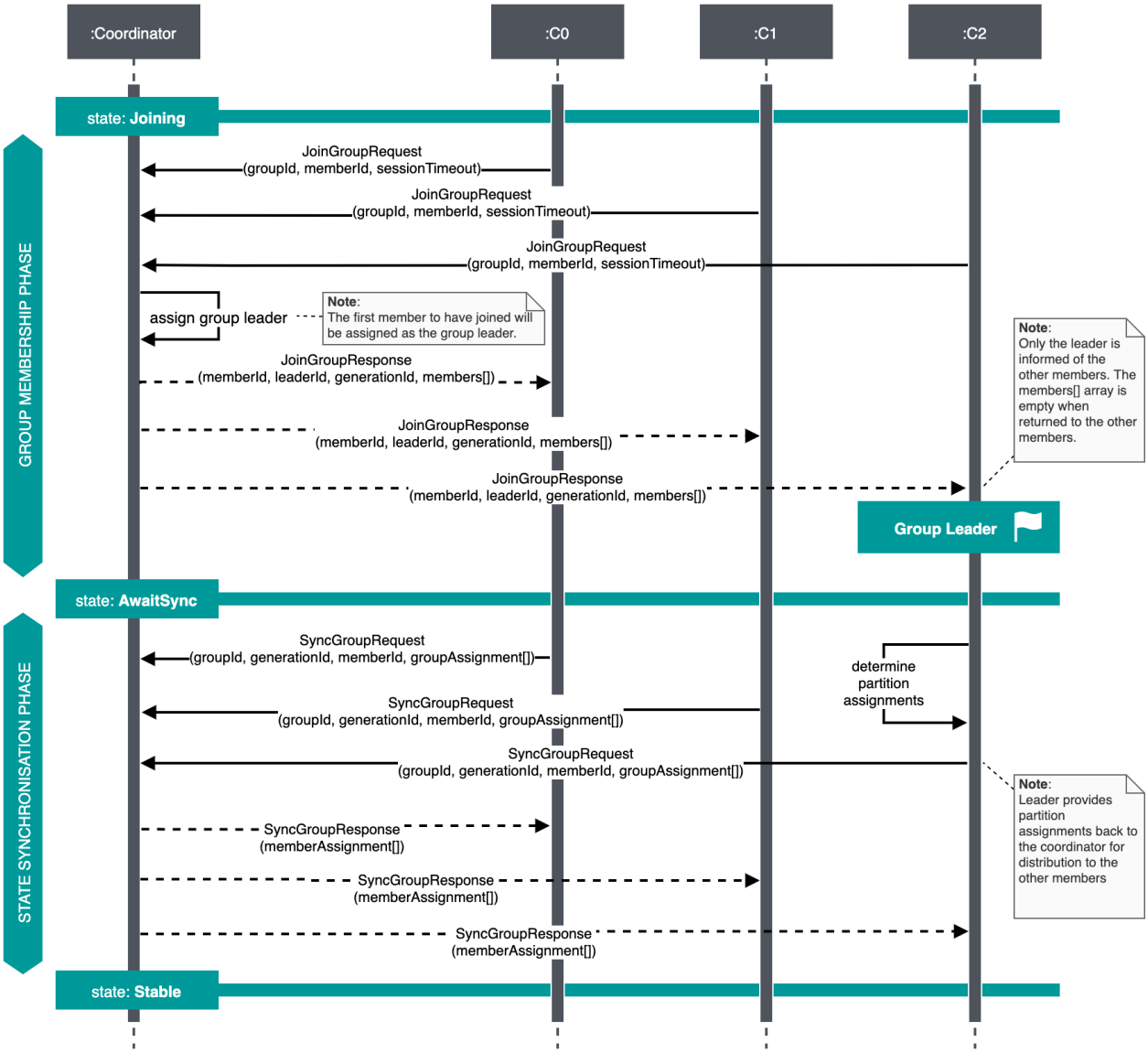
Group membership basics

A consumer group is a set of related consumers that contend for the assignment of a mutually exclusive set of partitions for the subscribed topics, such that a given topic-partition combination is assigned to *at most one* consumer. We say 'at most one' to cover the case when all consumers are offline or not responding. Rest assured — when all is well and there is at least one consumer in the group, a partition will be assigned to exactly one consumer.

Members of one consumer group cannot interfere with those of another group; in other words, consumer groups are completely isolated from one another.

Establishing group membership

Kafka's group management protocol is divided into two phases: *group membership* and *state synchronisation*. The group membership phase is used to identify the active members of the group and appoint a group leader. The state synchronisation phase is used by the group leader to derive the partition assignment for all group members, including itself, and to disseminate this state among the remaining population of consumers. The sequence diagram below illustrates the phases of the protocol. Explanations will follow.



Protocol phases

The group membership phase commences when a consumer sends a `JoinGroupRequest` to the group's coordinator. The request will contain the group ID (specified by the `group.id` configuration property), the internal member ID of the consumer (which is initially empty for new consumers), the desired session timeout (specified by `session.timeout.ms`), as well as the consumer's own metadata.

Upon receiving a `JoinGroupRequest`, the coordinator will update its state and select a leader from the group, if one has not been already assigned. The coordinator will park the request, intentionally delaying the response until all expected members of the group have sent their join requests. The delay acts as a synchronisation barrier, ensuring all members become simultaneously aware of one another. The responses will contain the set of individual member IDs and the ID of the group leader.

The coordinator of each group is implicitly selected from the partition leaders of the internal topic `__consumer_offsets`, which is used to store committed offsets. The group's ID is hashed to one of the partitions for the `__consumer_offsets` topic and the leader of that partition is taken to be the coordinator. By piggybacking on an existing leadership election process, Kafka conveniently avoids yet another arbitration process to elect group coordinators. Also, the management of consumer groups is divided approximately equally across all the brokers in the cluster, which allows the number of groups to scale by increasing the number of brokers.

State synchronisation

The responsibility of assigning topic-partitions to consumers falls upon the group leader, and is performed in the *state synchronisation* phase of the protocol. The assignment (sub-)protocol is conveniently tunnelled within the broader group management protocol and is concealed from the coordinator — in the sense that the coordinator largely treats the contents of state synchronisation messages as a 'black box'. As such, the assignment protocol is often referred to as an *embedded* protocol. Other embedded protocols may coexist, tunnelled in the same manner, potentially bearing no relation to partition assignment.

To make it abundantly clear, the terms 'group leader' and 'group coordinator' refer to different entities. The group leader is a consumer client that is responsible for performing partition assignment; the group coordinator is a broker that arbitrates group membership.

Once group membership has been established by the completion of the first phase, the coordinator will enter the `AwaitSync` state, waiting for the partition assignment to be performed on the leader. The leader will delegate to an implementation of a `ConsumerPartitionAssignor` to perform partition assignment, communicating the outcome of the assignment to the coordinator in a `SyncGroupRequest`.

In the meantime, all members will send their `SyncGroupRequest` messages to the coordinator. Having received the `SyncGroupRequest` from the leader, containing the partition assignments for each member, the coordinator replies with a `SyncGroupResponse` to all members of the group — informing them of their individual partition assignments and completing the protocol. Having done so, the coordinator enters the `Stable` state, and will maintain this state until group membership changes.

As of version 0.9.0.0, Kafka separates the management of group membership from partition assignment. While group arbitration is performed on the coordinator, the partition assignment is delegated to one of the members — the group leader. Before Kafka 0.9.0.0, the coordinator was responsible for both functions.

Delayed rebalance

Having the coordinator reply with a `JoinGroupResponse` immediately after all expected members have joined generally works well when a group is undergoing minor changes in population — for

example, as a result of scaling events or one-off consumer failures. However, if group members are joining at a rapid rate, the number of unnecessary rebalances will go through the roof. This 'swarm' effect can typically be felt on application startup or when previously connected clients reconnect *en masse* due to an intermittent network failure.

This issue was addressed in Kafka 0.11.0.0 as part of KIP-134[29]. A new broker property — `group.initial.rebalance.d` — was introduced for the group coordinator to allow additional time for consumers to join before sending `JoinGroupResponse` messages, thereby reducing the number of rebalances. The delay is only applied when the group is empty; membership changes in a non-empty group will not incur the initial rebalance delay. The default value of this property is `3000` (3 seconds).

A drawback of the initial rebalance delay is its treatment of a singleton group, where only one member exists. This is typical of test scenarios and local development. When using Kafka for small-scale testing or development, it is recommended that `group.initial.rebalance.delay.ms` is set to `0`.

Changing group membership

The coordinator maintains a generation ID — a monotonically-increasing 32-bit integer — corresponding to a point-in-time snapshot of the group membership state. When a new member wishes to join an existing group, they will send a `JoinGroupRequest` to the coordinator. Conversely, a member may gracefully leave the group at any point by sending a `LeaveGroupRequest`. Either action will result in the coordinator entering the `Joining` state, awaiting for *all* remaining members to rejoin the group.

While in the `Joining` state, the coordinator will reject heartbeats and other operations with a `REBALANCE_IN_PROGRESS` error. This will be detected by the remaining members upon their next heartbeat attempt, forcing them to rejoin with their existing member IDs. Once all members have successfully rejoined, the coordinator will increment the generation ID and transition to the `AwaitSync` state — initiating the synchronisation phase of the protocol and causing the rebalancing of partition assignments on the group leader. Upon successful rebalancing and state synchronisation, the coordinator will once again settle in the `Stable` state.

It may be possible for a member to miss a generational change in group membership — for example, due to a delayed heartbeat — and thereby be forcibly excluded from the group by the coordinator. The member might still legitimately believe it to be a part of the group and may attempt to fetch records or send commit requests. These members are referred to as 'zombies'. The use of an incrementing generation ID on the coordinator acts as a fencing mechanism — rejecting requests with an `ILLEGAL_GENERATION` error where the presented generation ID does not match the coordinator's generation ID. This forces zombie consumers to discard their assignments, clear their member ID, and rejoin the group from a clean slate.

[29] https://cwiki.apache.org/confluence/display/KAFKA/KIP-134%3A+Delay+initial+consumer+group+rebalance

State synchronisation barrier

When transitioning partition assignments from one consumer to another — following a member join or leave event — the consumers must collectively ensure that there is no time-overlap between the assignment of a partition to a new consumer and the release of the partition from its previous holder. While the time-separation between the revocation and the assignment is undesirable from a performance standpoint, allowing the two events to overlap in time is unacceptable. Therefore, we must conclude that some degree of separation is necessary and acceptable.

To achieve this separation, a consumer client must inform its user of any impending changes to the partition assignments, which is done through an `org.apache.kafka.clients.consumer.ConsumerRebalanceListene` callback interface, listed below sans the Javadoc comments.

```
public interface ConsumerRebalanceListener {
  void onPartitionsRevoked(Collection<TopicPartition> partitions);

  void onPartitionsAssigned(Collection<TopicPartition> partitions);

  default void onPartitionsLost(
      Collection<TopicPartition> partitions) {
    onPartitionsRevoked(partitions);
  }
}
```

A rebalance listener is registered with a `KafkaConsumer` client when calling one of its overloaded `subscribe()` methods.

After completing the group join, and just before initiating the state synchronisation phase, a consumer will assume that all partitions that it presently holds will be revoked by the leader, and will invoke the `onPartitionsRevoked()` callback accordingly, passing it the complete set of partitions that the consumer had owned prior to the rebalance. The execution of the callback may block for as long as necessary, waiting for the application to complete any in-flight activities. Only when the callback completes, will the client send a `SyncGroupRequest` message to the coordinator. (The synchronisation behaviour is different when incremental cooperative rebalancing is enabled, which will be discussed shortly.)

Later, upon completion of the state synchronisation phase, the consumer will learn of its updated assignments and will invoke `ConsumerRebalanceListener.onPartitionsAssigned()` with the partitions that the consumer now owns. (This may include partitions it previously owned if they haven't been reassigned.) The `onPartitionsRevoked()` method effectively acts as a barrier — satisfying the strict requirement that the `onPartitionsRevoked()` callback is invoked on *all* consumers before `onPartitionsAssigned()` is invoked on *any* consumer.

Without this barrier in place, it would have been possible for the new assignee of a partition to commence the processing of records before the outgoing consumer has had a chance to relinquish

control over that partition. The barrier might result in a short period of time where none of the partitions are being processed during the rebalancing phase. Kafka's design documentation colloquially refers to this as a *stop-the-world* effect, akin to its namesake in the area of garbage collection (GC) algorithms. Indeed, a stop-the-world rebalance in Kafka has the same drastic effect on the throughput of consumers as the GC has on the application as a whole. And unlike GC, which affects one process at a time, a stop-the-world rebalance affects all consumers simultaneously.

The final callback method — `onPartitionLost()` — signals to the application the loss of prior partitions due to a forced change of ownership. This will happen when a consumer fails to emit a heartbeat within a set deadline, thereby being considered 'dead' by the coordinator. The implications of partition loss are described in detail in the subsequent section on *liveness and safety*.

Incremental cooperative rebalancing

One of the main drawbacks of the original state synchronisation protocol, also known as the *eager rebalancing protocol*, is that it only allocates one phase to the entire rebalancing operation, which is one request-response message exchange in practical terms. This results in consumers taking an overly pessimistic view on the revoked partitions — forced to assume that all partitions might be revoked — whereas in reality only a subset of the partitions will typically be reassigned following a rebalance. This unnecessarily triggers the consumers to clean up all partitions, extending the duration of the rebalancing phase, and therefore, the extent of the stop-the-world pause.

Incremental cooperative rebalancing is a recent improvement of the state synchronisation protocol, introduced in Kafka 2.4.0 under KIP-429[30]. Under the cooperative model, the join and rebalance phases may be repeated, each introducing an incremental change to partition assignment. The cooperative rebalancing protocol introduces a second, follow-up join-rebalance round immediately following the first join-rebalance.

Cooperative rebalancing adds a new rebalance protocol version, which is referred to by the `COOPERATIVE` constant (`0x01` byte representation), in contrast to the original `EAGER` constant (`0x00`). The use of cooperative rebalancing is optional, enabled automatically in response to the chosen rebalancing strategy. (Rebalancing strategies will be discussed in one of the subsequent sections.) When the cooperative protocol is in force, the consumer does not invoke the `onPartitionsRevoked()` callback before sending the `SyncGroupRequest`; instead, the callback is deferred to such time when the `SyncGroupResponse` is received, and only for a non-empty set of revoked partitions. The client will then invoke the `onPartitionsAssigned()` callback, passing it the set of newly assigned partitions, even if that is an empty set. Having processed the callbacks, the client will initiate a second rejoin, followed by a subsequent rebalance.

The trick to making this work is to separate the revocations from the assignments by exactly one round, so that the first join-rebalance round comprises exclusively of revocations, or of assignments where no prior assignees exist (as in the case of the initial join). The second join-rebalance round comprises only of new assignments where a revocation was communicated in the previous round.

[30] https://cwiki.apache.org/confluence/display/KAFKA/KIP-429%3A+Kafka+Consumer+Incremental+Rebalance+Protocol

Thus, for any given partition, the `onPartitionsRevoked()` callback will always complete on the outgoing assignee before the `onPartitionsAssigned()` callback is instigated on the new assignee.

Between the two rounds, the group is said to be in an *unbalanced* state — not all partitions may be assigned to a consumer, even if there are sufficient members in the group. By contrast, the eager rebalance protocol will ensure that a partition is always assigned to a consumer at the conclusion of a synchronisation round, providing that at least one group member exists. The unbalanced state is resolved upon the completion of the second join-rebalance round.

The main benefit of cooperative rebalancing is in communicating the *exact* revocations to consumers, rather than coercing them towards a worst-case assumption. By reducing the amount of cleanup work consumers must perform during the `onPartitionsRevoked()` callback, the duration of the rebalancing stage is reduced, as is the effect of the stop-the-world pause.

A secondary benefit is the marked improvement of rebalancing performance and reduced disruptions to consumers in the event of a *rolling bounce* — where a rolling restart of a consumer population results in repeat leave, join, and state synchronisation events. Rebalancing still occurs, but the impact of this is greatly reduced as only the bounced clients' partitions get shuffled around.

> In theory, the protocol may be extended in the future to accommodate an arbitrary number of join-rebalance rounds and may even space them out if necessary. In the (hypothetical) extended protocol, the balanced state would be incrementally converged on after the completion of *N* rounds.
>
> Another prospective enhancement would be the addition of a 'scheduled rebalance timeout', granting temporarily departing consumers a grace period within which they can rejoin without sacrificing prior partition assignments. This is meant to combat the full effects of a rolling bounce by avoiding revocations.
>
> In addition, a scheduled rebalance timeout could deal with self-healing consumers, where the failure of a consumer is detected by a separate orchestrator and the consumer is subsequently restarted. (Kubernetes is often quoted as a stand-in for that role.) With a sufficiently long timeout, the orchestrator can detect the failure of a consumer process and restart its container without penalising the process by way of revoking its partitions. This allows the failed process to be restored faster and also minimises the impact on the healthy consumers. (Similar characteristics can be achieved using the existing static membership protocol, as we will shortly see.)

The main challenge of the cooperative model is the added join-rebalance round, leading to more network round-trips. For cooperative rebalancing to be effective, the underlying partition assignment strategy must be *sticky* — in other words, it must preserve the prior partition assignments as much as possible. Without a sticky rebalancing strategy, the cooperative model reduces to the eager rebalancing model, albeit with more overhead and complexity.

A word of caution: Enabling the cooperative protocol changes the semantics of the `ConsumerRebalanceListener` callback. Under the eager rebalance protocol, the `onPartitionsAssigned()` is handed the complete set of assigned partitions; whereas, under the cooperative protocol, the `onPartitionsAssigned()` is given just the set of new partitions that were acquired since the last rebalance.

Static membership

Static group membership takes a different approach to the conventional (join-triggered) partition assignment model by associating group members with a long-term, stable identity. Under the static model, members are allowed to leave and join a group without forfeiting their partition assignment or causing a rebalance, provided they are not away for longer than a set amount of time.

The main rationale behind static membership is twofold:

1. To reduce the impact of rebalancing and the associated interruptions when *inelastic* consumers bounce; and
2. To allow for an external health checking and healing mechanism for ensuring liveness of the consumer ecosystem.

The second point will be discussed later in this chapter, in the broader context of *liveness* and *safety* concerns. For the time being, we shall focus on the first point.

An 'inelastic' consumer is one that is expected to persist for an extended period of time, occasionally going down for a short while — for example, to perform a planned software deployment or as a result of an intermittent failure. In both cases, certain assumptions might be made as to the duration of the outage. The population of inelastic consumers tends to be more or less fixed in size. By contrast, an 'elastic' consumer is likely to be spawned in response to an elevated load demand and will be terminated when the demand tapers off.

Static assignment works by making minor amendments to the group management and state synchronisation protocol phases. It also adds a property — `group.instance.id` — to the consumer configuration, being a free-form stable identifier for the consumer instance.

Upon joining the group, a static member will submit its member ID in a `JoinGroupRequest`, along with the group instance ID in the member metadata. If the member ID is empty — indicating an initial join — the coordinator will issue a new member ID, as per the dynamic membership scenario. In addition to issuing a member ID, the coordinator will take note of the issued ID, associating it locally with the member's group instance ID. Subsequent joins with an existing member ID will proceed as per the dynamic group membership protocol.

Having joined the group, the consumer will impart its group instance ID to the group leader, via the coordinator. Like dynamic members, static members will have a different member ID on each

join, rendering it largely useless for correlating partition assignments across generations. The built-in assignor implementations have added provisions for this — using the group instance ID, where one is supplied, in place of the member ID.

Unlike its dynamic counterpart, a static consumer leaving the group will not send a `LeaveGroupRequest` to the coordinator. When the consumer eventually starts up, it will simply join with a new member ID, as per the explanation given above. The coordinator will still enforce heartbeats as per the consumer-specified `session.timeout.ms` property. If a bounced consumer fails to reappear within the heartbeat deadline, it will be expunged from the group, triggering a rebalance. In other words, there is no difference in the health check behaviour between the static and dynamic protocols from the coordinator's perspective. The session timeout is typically set to a value that is significantly higher than that for an equivalent dynamic consumer. The reasons for this will be discussed later.

With the recent introduction of incremental cooperative rebalancing and the support for static group membership, it is likely that the `group.initial.rebalance.delay.ms` property will be deprecated in the near future.

One of the challenges with the static group membership model is the added administrative overhead — namely, the need to provision unique group instance IDs to each partaking consumer. While this could be done manually if need be, ideally, it is performed automatically at deployment time. At any rate, a blunder in the manual process or a defect in the deployment pipeline might lead to a situation where two or more consumers end up with identical group instance IDs. This begs the question: will two identically-configured static consumers lead to a non-exclusive partition assignment?

Kafka deals with the prospect of misconfigured consumers by implementing an internal fencing mechanism. In addition to disclosing its group instance ID in the group membership phase, the consumer will also convey this information in `SyncGroupRequest`, `HeartbeatRequest`, and `OffsetCommitRequest` messages. The group coordinator will compare the disclosed group instance ID with the one on record, raising a `FENCED_INSTANCE_ID` error if the corresponding member IDs don't match, which propagates to the client in the form of a `FencedInstanceIdException`.

Fencing is best explained with an example. Suppose consumers *C0* and *C1* have been misconfigured to share an instance ID *I0*. Upon its initial join, *C0* presents an empty member ID and is assigned *M0* by the coordinator. The coordinator records the mapping *I0* ⟶ *M0*. *C1* then does the same; the coordinator assigns *M1* to *C1* and updates the internal mapping to *I0* ⟶ *M1*. When *C0* later attempts to sync with the *I0*-*M0* tuple, the coordinator will identify the offending mapping and respond with a `FENCED_INSTANCE_ID` error.

If *C0* completes both a join and a synchronisation request before *C1* does its join, then the inconsistency will be identified during the synchronisation barrier, just prior to the coordinator replying with a `SyncGroupResponse`. Either way, one of the consumers will be fenced.

Liveness and safety

It would be rather nice if consumers and brokers never failed, networks were reliable, and group membership changes were always cleanly demarcated by join and leave requests. Unfortunately, this cannot be; we have to contend with the bleak harshness of reality, which is particularly exacerbated when dealing with distributed systems.

In the realm of distributed systems, like in concurrent computing, there are two fundamental properties of interest: *liveness* and *safety*.

Liveness is a property of a system that requires it to make progress despite its internal components competing for shared resources. A system that exhibits liveness will guarantee that something 'good' will *eventually* happen.

Safety is an orthogonal property that guarantees that none of the critical invariants of a system will ever be violated. In other words, something 'bad' will *never* happen under a system that exhibits safety.

The 18th century English jurist William Blackstone expressed a formulation that has since become a staple of legal thinking in Anglo-Saxon jurisdictions. Blackstone's formulation was: *It is better that ten guilty persons escape than that one innocent suffer.*

Sir William Blackstone (1723 – 1780)

John Adams — the second president of the United States — remarked on Blackstone's formulation that, paraphrasing: *Guilt and crimes are so frequent, that all of them cannot be punished, whereas*

the sanctity of innocence must be protected. One way of looking at this remark, although it was not conveyed verbatim, is that the guilty will likely reoffend, and therefore will likely be caught eventually. Conversely, if the innocent were to lose faith in their security, they would be more likely to offend, for *virtue itself is no security*, to use Adams's own words.

John Adams (1735 — 1826)

Unifying Blackstone's formulation with Adams's, we can derive the following: *The guilty should be punished eventually, while the innocent must never be punished.* How does this relate to our previous discussion on distributed systems? As it happens, the aforementioned statement combines both *liveness* and *safety* properties. Enacting of punishment for the guilty, at some indeterminate

> point in time, is a manifestation of *liveness*. Ensuring that the innocent are never punished is *safety*.
>
> Interestingly, the liveness property may not be satisfied in a finite execution of a distributed system because the 'good' event might only theoretically occur at some time in the future. Eventual consistency is an example of a liveness property. Returning to the Blackstone-Adams formulation, there is no requirement that the guilty are caught within their lifetime, or ever, for that matter.
>
> What is the point of a property if its cardinal outcome may never be satisfied? Formally, the liveness property ensures that an eventual transition to a desirable state is theoretically possible from every conceivable state of the system; in other words, there is no state which categorically precludes the 'good' event from subsequently occurring. Theoretically, there is no crime that cannot be solved.
>
> By comparison, the safety property can be violated in a finite execution of a system. Just one occurrence of a 'bad' event is sufficient to void the safety property.

Kafka gives us its assurance, that providing at least one consumer process is available, then all records will eventually be processed in the order they were published, and no record will be delivered simultaneously to two or more competing consumers. Granted, in reality, this guarantee only holds if you use Kafka correctly. We have witnessed numerous examples in Chapter 10: Client Configuration where a trivially misconfigured, or even a naively-accepted default property will have a drastic impact on the correctness of a system. Let us look the other way for the moment, blissfully pretending that clients have been correctly configured and the applications are defect-free.

How does all this relate to liveness and safety? The preservation of record order and the assignment of any given partition to at most one consumer is a manifestation of the *safety* property. The eventual identification of a failed consumer and the prompt rebalancing of partition assignments takes care of *liveness*.

Kafka satisfies the liveness property in two ways:

1. **Checking availability** — by requiring consumers to periodically exchange heartbeats with the group coordinator, thereby indicating that the consumer process is running on the host and that a network path is available between the consumer process and the coordinator node.
2. **Checking progress** — by requiring that the consumer periodically invoke `poll()`, thereby indicating that it is able to handle its share of the partition assignment.

The consumer will send a `HeartbeatRequest` message at a fixed interval from a dedicated heartbeat thread. The coordinator maintains timers for each consumer, resetting the clock when a heartbeat is received. Having received the heartbeat, the coordinator will reply with a `HeartbeatResponse`. Conversely, if the coordinator does not receive a heartbeat within the time specified by the `session.timeout.ms` consumer property, it will assume that the consumer process has died. The coordinator will then transition to the `Joining` state, causing a rejoin of all remaining members. While the session timeout is configured on the consumer, it is communicated to the coordinator inside a `JoinGroupRequest` message, with the deadline subsequently enforced by the coordinator.

The `session.timeout.ms` property defaults to `10000` (10 seconds). It is recommended that this value is at least three times greater than the value of `heartbeat.interval.ms`, which defaults to `3000` (3 seconds). As a further constraint, the value of `session.timeout.ms` must be in the range specified by the `group.min.session.timeout.ms` and `group.max.session.timeout.ms` broker properties, being `6000` (6 seconds) and `1800000` (30 minutes), respectively, by default. By binding the range of the session timeout values on the broker, one can be sure that no client will connect with its `session.timeout.ms` setting outside the permitted range.

While the availability check is performed on the coordinator node, the progress check is performed locally on the consumer. The background heartbeat thread will maintain its periodic heartbeating for long as the last recorded poll time is within the deadline specified by the `max.poll.interval.ms` property, which defaults to `300000` (5 minutes). If the application fails to invoke `KafkaConsumer.poll()` within the specified deadline, the heartbeat thread will cease, and a `LeaveGroupRequest` will be sent to the coordinator. Stated otherwise, the heartbeat thread doubles up as a watchdog timer — as soon as the poll deadline is missed, the client internally 'self-destructs', causing a leave, followed by a rejoin.

The client should assign `session.timeout.ms` and `heartbeat.interval.ms` values based on the application's appetite for failure detection. Shorter heartbeat timeouts result in quicker failure detection at the cost of more frequent consumer heartbeating, which can overwhelm broker resources and lead to false-positive failure events — whereby a live consumer is mistakingly declared dead. Longer timeouts lead to less sensitive failure detection, but may leave some partitions unhandled in the event of consumer failure.

A certain degree of process predictability and some amount of slack in the session timeout are required for a consumer to consistently meet its heartbeat deadline — the *"prove to me that you are still alive"* check. Given a suitably reliable network and unsaturated consumer processes with regular garbage collector (GC) activity, the heartbeating process can be made to work reliably and predictably — even in the absence of a genuinely deterministic runtime environment. In the author's experience, the default values for the heartbeat interval and the session timeout are sufficient in most scenarios, occasionally requiring adjustments to compensate for heavily loaded consumer applications or network congestion.

The situation with the progress check — the *"prove to me that you are still consuming records"* audit — is somewhat more challenging, as it relies on a round of polling to complete within a bounded time. Depending on what the consumer does with each record, there is little one can do to ensure that the processing of records will unconditionally complete within a set timeframe.

When faced with nondeterministic record processing, one way of improving the situation is to cap the number of records that will be returned to the application by `KafkaConsumer.poll()`. This is controlled by the `max.poll.records` consumer property, which defaults to `500`. This doesn't change the way records are fetched from the broker, nor their quantity; instead, the `poll()` method artificially limits the number of records returned to the caller. The excluded records will remain stashed in the fetch buffer — to be returned in a subsequent call to `poll()`.

In reducing the number of records returned from `poll()`, the application effectively slackens its

processing obligations between successive polls, increasing the likelihood that a cycle will complete before the `max.poll.interval.ms` deadline elapses, particularly if the average and worst-case processing times of a record are well known.

One might think that liveness and safety only apply to niche areas — low-level systems programming, operating systems, and distributed consensus protocols. In actual fact, we are never completely insulated from these properties — we might just experience their effects in different and often subtle ways, sometimes barely realising it.

Dealing with failures

Whilst an average (mean or median) processing time can often be derived empirically, the distribution of processing times may, and often does, exhibit a 'long tail'. In other words, the worst-case processing time might not be bounded. To illustrate this, consider a fairly typical event streaming example where the processing of a record results in the updating of a database or perhaps invoking some downstream service. These sorts of operations are inherently fault-prone and may time out, requiring retries. In the worst-case scenario, a downstream dependency might be experiencing downtime, leaving the consumer in an indefinite retry loop. Clearly, even with one backlogged record, the consumer might not make the `max.poll.interval.ms` deadline. So what to do?

Unfortunately, Kafka does not have an answer of its own. The user is left to fend for themselves. As it happens, there are five strategies, at least, for dealing with this:

1. Set an absurdly large `max.poll.interval.ms` value, to effectively disable the progress check.
2. Maintain a reasonable progress deadline, allowing Kafka to detect progress failure and rebalance the group.
3. Detect an impending `max.poll.interval.ms` deadline and voluntarily relinquish the subscription at the consumer.
4. Implement a record-level deadline within the consumer, such that if a record fails to make progress within a set timeframe, it is recirculated back to the tail-end of the original topic.
5. Implement a record-level deadline; the record will be skipped if it is unable to make the deadline, and, ideally sent to a dead-letter topic for subsequent post-mortem.

The first option takes the stance of *"do it 'till it's done, whatever it takes"*. This may be appropriate if the correctness of the application depends on having every record processed and in the order that the records appear in the topic. We are effectively saying that there is no point progressing if the process cannot do its job, so it might as well stall until the downstream issue is rectified.

The main drawback of this approach is that it assumes that failure to make progress can only be attributed to an external cause, such as the failure of a downstream dependency. It fails to take into account potential failures on the consumer, such as a software defect that may have caused a deadlock, or some other *local* contingency that is impeding progress.

The second option yields a similar outcome to the first, in that it will not progress the consumption of records on the partition containing the troublesome record. The progress check is still in force, meaning that the poll loop will time out, and the coordinator will exclude the consumer from the group, triggering a partition rebalance. The new assignee of the troublesome partition will presumably experience the same issue as the outgoing consumer, and time out in the same manner. The previous assignee would have rejoined the group, and, in doing so, will have become eligible for another share of partition assignments. This cycle will continue until the downstream issue is resolved and the consumers are eventually able to make progress. The advantage of this approach is that it makes no assumptions as to the cause of the problem. If the progress was originally impeded by a deadlocked consumer, then the new consumer should proceed without a hitch. It also requires no additional complexity on the consumer's behalf.

This approach is idiomatic to Kafka, but it is not without its drawbacks. By rebalancing the partitions and letting the new assignee time out, the overall throughput of the topic may be impacted, causing periodic stutter. Another, more serious drawback, is that although Kafka will detect consumer failure, the consumer itself might not. Kafka will transfer ownership of the impacted partitions to the remaining consumers, while the outgoing consumer might assume that it still holds those partitions and will slowly work through its backlog. This problem is explored in more detail in the section titled *'Dealing with partition exclusivity'*.

The third strategy — voluntarily relinquishing a subscription — is an evolution of the second approach, adding timing logic and progress checks to the poll-process loop, so that the consumer process can preempt an impending deadline failure. This would involve proactively cleaning up its state and unsubscribing. The preemption mechanism implies either some form of non-blocking processing of records or the ability to interrupt the poll-process thread so that it does not block indefinitely. Once a subscription is forfeited, the consumer immediately resubscribes — resetting its state within the group.

The benefit of this approach is that it maintains the strongest safety guarantees with respect to record ordering and at-least-once delivery — much like the first two approaches. Akin to the second strategy, it satisfies the liveness property by proactively yielding its subscription, while still allowing the coordinator to detect consumer failures. This strategy has the added safety advantage of enforcing record processing exclusivity by adding a conservative local failure detection mechanism, ensuring that record processing does not overrun.

A minor drawback of this model is that it causes a rebalance as part of forfeiting the subscription, shortly followed by a second rebalance as the consumer rejoins the pack. Each rebalance will disrupt *all* consumers, albeit for a short time. In Kafka parlance, this is referred to as 'bouncing' — whereby consumers come in and out of the group, causing no net effect, but forcing an unnecessary reshuffling of partition assignments. To be fair, the behaviour of this strategy with respect to rebalancing is no worse than strategy #2. Note that the effect of rebalancing under this strategy cannot be ameliorated by employing a static membership model because unsubscribing from a topic will cause an implicit rebalance.

The fourth approach — implementing a record-level deadline — essentially involves starting a timer just before processing a record and ensuring that no operation blocks beyond the processing deadline.

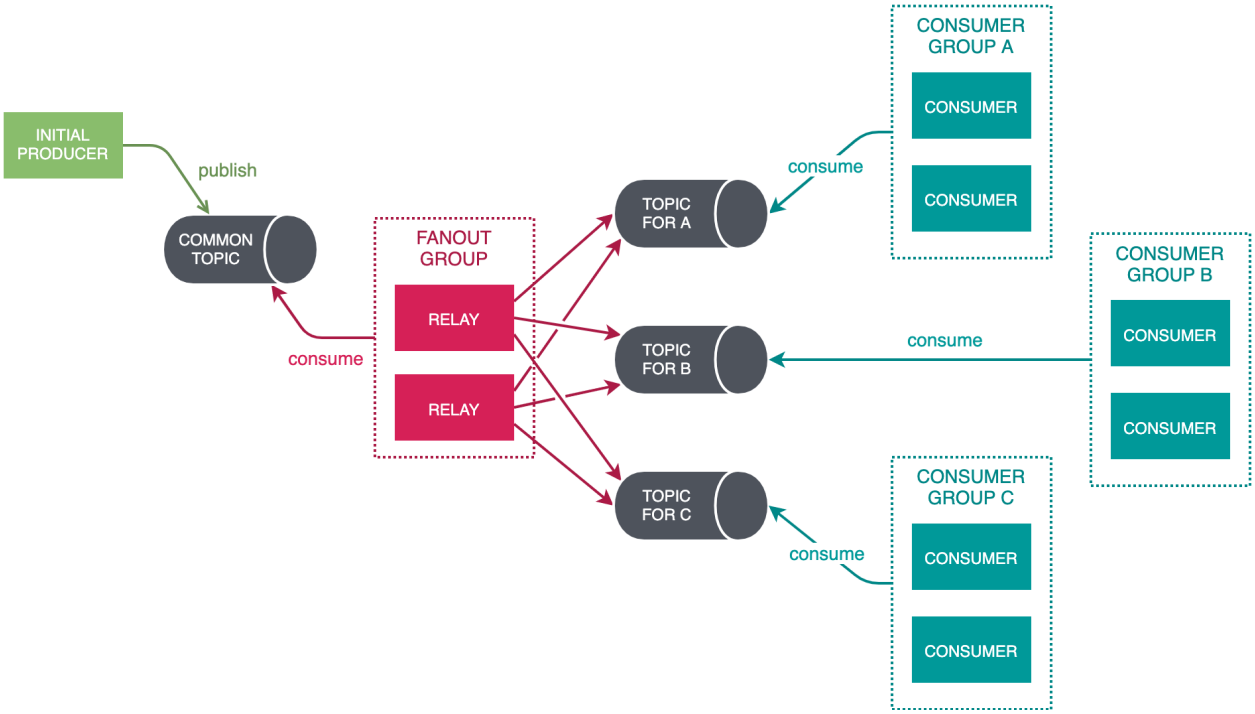
When this deadline is reached, the consumer takes corrective action. In this variant, the corrective action involves republishing the record to its original topic, such that it will be dealt with *eventually* by some consumer — the consumer that republished the record or some other consumer if the topic's partitions were reassigned between re-queuing the record and its subsequent consumption.

The record-level timeout must be significantly shorter than `max.poll.interval.ms` for it to be effective. In fact, it should be just under the quotient of dividing `max.poll.interval.ms` by `max.poll.records` — allowing for the worst-case scenario for all records in the batch. (*Worse than worst*, if such a phrase is grammatically admissible.)

This model is predicated on the consumers' capacity to arbitrarily reorder records; in other words, it only works if preserving the original record is not essential to the correct operation of the system. This is typically not the case in most event streaming scenarios, but it may be suitable when records are entirely self-contained and independent of one another, or when the records can be trivially reordered on the consumer. It should also work well in more traditional peer-to-peer message queuing scenarios, where messages might correspond to discrete tasks that can be processed out of order. Consider, for example, a queue-based video transcoding application, where each record is a pointer to a media file stored in an object store (such as AWS S3) along with instructions for the target format, resolution, bitrate, etc. Now, suppose the consumer-end of the transcoder requires a licensed codec for a specific video, and is experiencing a problem with a downstream license server. It would be reasonable for the transcoder to re-queue the record, leaving it in the queue until such time that the license server is back online.

There are several potential variations of the strategy above. The producer might include an expiry time in the record indicating how long the task should be in a queue before it is deemed useless to the business. The consumer would only attempt to re-queue the record for as long as it hasn't outlived its useful lifespan. In another variation, the consumer may want to limit the number of retry attempts. This can be accomplished by setting a retry counter when the record is first published, validating and decrementing it on each subsequent re-queuing attempt. When the record no longer qualifies for re-queuing, it will be discarded. The consumer might also wish to log this fact and publish a copy of the discarded record to a dead-letter topic.

Republishing records mutates the topic for *all* consumer groups. As such, reordering strategies are only effective for point-to-point messaging scenarios, involving just one consumer group. In order to apply a reordering strategy to (multi)point-to-multipoint scenarios, where multiple consumer groups share a source topic, it is necessary to implement a *fan-out* topology. This involves transforming a (multi)point-to-multipoint topic into a (multi)point-to-point topic, coupled to several point-to-point topics — one for each consumer group. The fan-out model eliminates sharing — each consumer group gets a private topic which it may mutate as it chooses.



Topic fanout

The final strategy is a limiting case of strategy #4, where the retry attempts counter is set to zero. In other words, a record that fails to complete within the allotted deadline is discarded immediately, with no second-chance re-queuing. This is the most relaxed model, which guarantees progress under a significantly relaxed notion of safety.

Dealing with partition exclusivity

If was previously mentioned that Kafka's assurance extends to the exclusive processing of records; namely, that *at most one consumer will be allocated to a partition, for any given consumer group.*

There is a subtle caveat here, albeit a crucial one — concealed in the wording of the statement above. The assurance is given only with respect to partition assignment; it doesn't cover the act of concurrently processing the record, despite what most Kafka practitioners might like to believe.

Specifically, the problem is this: consider two consumers *C0* and *C1* in a common group, contending over one partition *P*. *P* is initially assigned to *C0*, and the group coordinator has a stable view of the group membership, being the set containing *C0* and *C1*.

At some point in the course of processing records, *C0* is unable to satisfy its progress check while working through a batch of records. Furthermore, *C0* is blocked while processing one of the records in the batch, with more records remaining. The `KafkaConsumer` instance that is backing *C0* will detect that the application has failed to invoke `poll()` within the progress deadline, and the heartbeat thread will consequently send an explicit leave request. However, the consumer remains blocked on whatever operation it was doing, oblivious to the background goings-on of the heartbeat thread.

Having received the leave request, the coordinator will adjust the group membership to a singleton set comprising *C1*, which will be followed by a partition reassignment. It is possible that *C0* may have rejoined the group, but that hardly matters — *P* has been reassigned to *C1*. When *C0* eventually unblocks, it will proceed with its backlog, naively assuming that it is still the owner of *P*. In the worst-case scenario, *C0* could enter a critical section, where the effect of processing a record on *C0* may conflict with the concurrent or earlier processing of an identical record on *C1*.

This demonstrates that although the partition was assigned exclusively from the leader's perspective, the restrictions under this assignment were not reflected commensurately on the consumers. Under normal circumstances, when the consumer population changes, the new and existing consumers are notified of changes to their partition assignments via an optional `ConsumerRebalanceListener` callback. The callback, which will be discussed in detail later, essentially informs a consumer that its existing partitions were revoked and that new partitions were assigned. Crucially, the callback is globally blocking in the revocation phase — no consumer is allowed to proceed with their new partition assignment until *all* consumers have handled the revocation event. This gives a consumer a vital opportunity to complete the processing of any in-flight records, gracefully handing its workload over to the new consumer. However, *the callback is executed in the context of the polling thread*, during its next call to `KafkaConsumer.poll()`. In our earlier example, *C0* was blocked — clearly, there was no way it could have called `poll()`. Furthermore, as it was forcibly excluded from the group, the consumer would not have been eligible to partake in the blocking revocation callback; a member that is deemed dead is not able to affect the remaining population of the group. (Its abilities will be restored when it rejoins the group, by which time the group would have undergone at least one rebalance.)

Kafka 2.4.0 introduces another `ConsumerRebalanceListener` callback method — `onPartitionsLost()` — designed to indicate to the consumer that its partitions have been forcibly revoked. This method is a best-effort indicator only; unlike the `onPartitionRevoked()` method, it does not act as a synchronisation barrier — it cannot stop or undo the effects of the revocation, as the new assignee will likely have started processing records by the time the outgoing consumer has reacted to `onPartitionsLost()`. Furthermore, like its peer callback methods, `onPartitionsLost()` executes in the context of the polling thread, invoked from within a call to `KafkaConsumer.poll()`. Since the failure scenarios assume that the consumer may be blocked indefinitely, the usefulness of this callback is questionable. Still, at least now Kafka informs us of the calamity that ensued, whereas in previous versions the consumer had to discover this of its own accord. More often than not, the effects of two consumers operating on the same records were discovered by disheartened users of the system, sometimes long after the fact.

By default, and to maintain compatibility with pre-2.4.0 consumer applications, the `onPartitionsLost()` callback simply delegates to `onPartitionRevoked()`. However, the roles of these callback methods are vastly different. When upgrading a consumer application that implements the `ConsumerRebalanceListener` callback to utilise version 2.4.0 (or newer) of the client library, one should immediately override the default implementation.

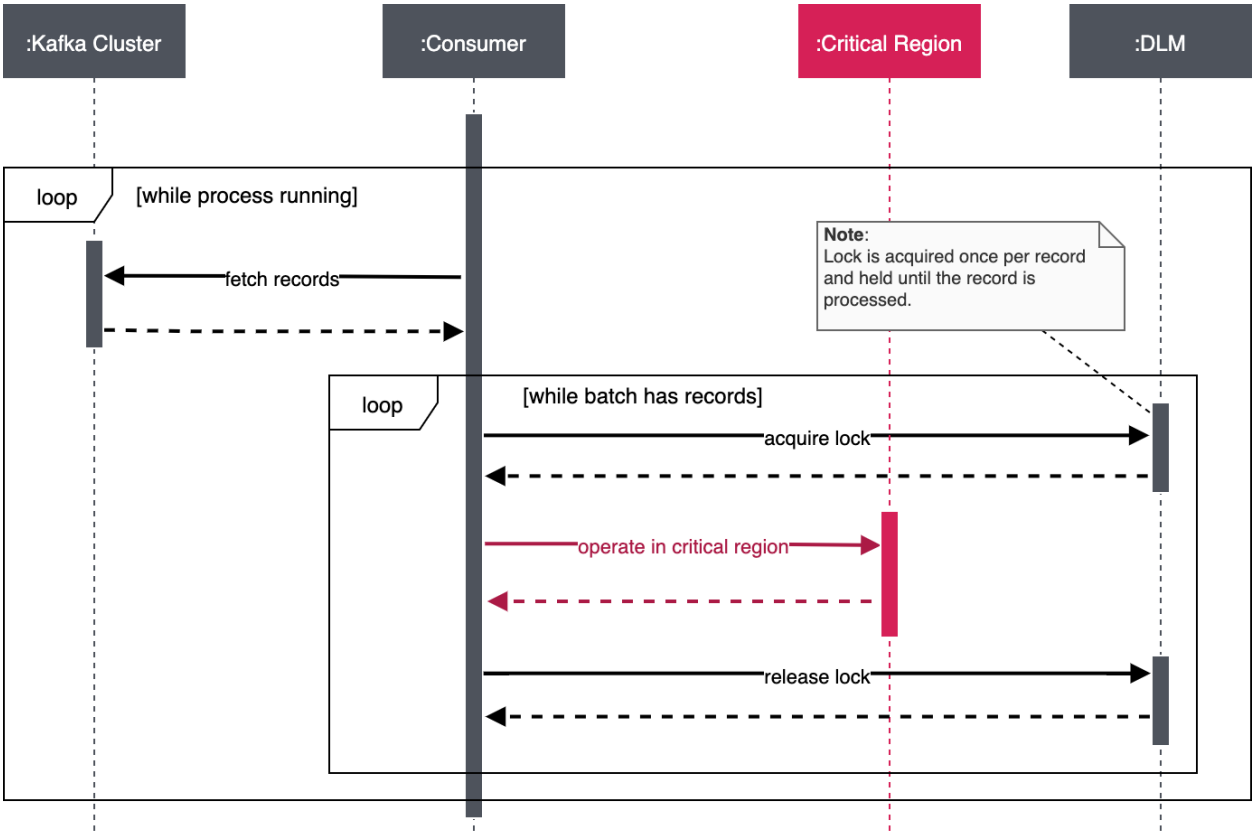
So how does one prevent the non-exclusive assignment condition? Again, Kafka is coy on answers.

Once more, we revert to outside strategies for dealing with partition exclusivity. There are three at our disposal:

1. Ensure that the poll loop always completes within the progress deadline set by `max.poll.interval.ms`, thereby sidestepping the problem.
2. Employ an external fencing mechanism for dealing with critical sections.
3. Employ fencing at the process level, terminating the outgoing process, or isolating it from the outside world. This may be used in addition to #2.

The first strategy has already been covered in the previous section. It tries to avoid a deadline overrun by either yielding the subscription, shedding the load, or requeueing records. It is not a universal strategy, in that it has drawbacks and might not apply in all cases.

The second strategy relies on an external arbitration mechanism — typically a distributed lock manager (DLM) — used to ensure exclusivity over a critical section. Before processing a record, the consumer process would attempt to acquire a named lock with the DLM, proceeding only upon successful acquisition. The name of the lock can be derived from the record's key or the partition number. This aligns critical sections to partitions and ensures that only one consumer may operate within a critical section at any one time; however, it does not preclude the record from being processed multiple times, as the two consumers may have handled the record at different, non-overlapping times. This is illustrated in the sequence diagram below.



Critical region — per-record

Having entered the critical section, the consumer will check that the effects of the record haven't already been applied by some other process. If it detects that the record has already been processed elsewhere, the consumer can safely skip the record and move on to the next.

In addition to promoting safety, the check for prior processing adds *idempotence* to record consumption — ensuring that the repeat processing of a record leads to no observable effects beyond those that were emitted during the first processing.

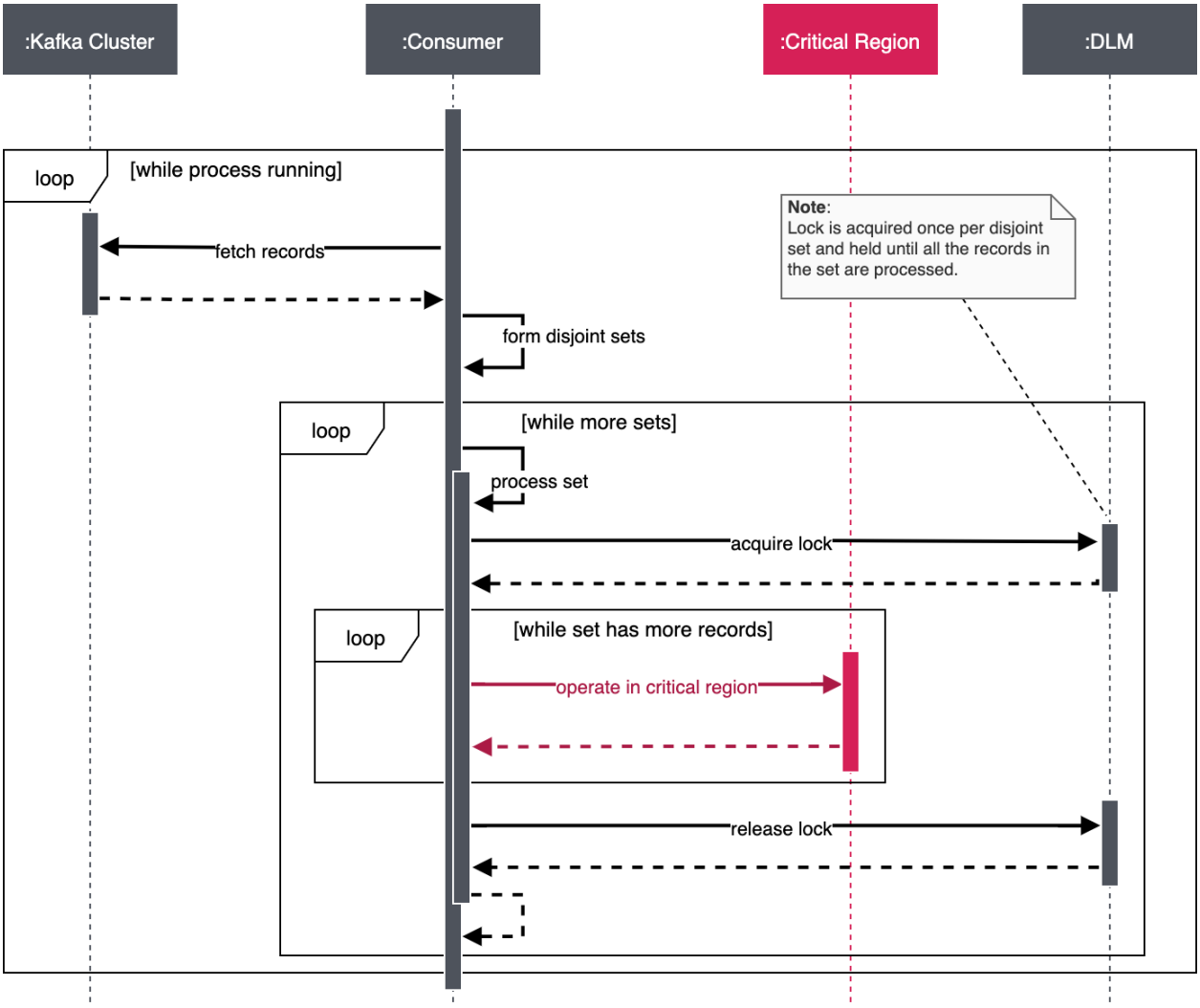
While traversing the critical section, a consumer's observation that a record has already been processed might lead to the following deduction. If one record has already been processed elsewhere, then it is likely due to a breach of the exclusivity constraint. Consequently, it is likely that multiple records have been processed elsewhere.

This deduction might lead to a tempting optimisation. Rather than trying the next record in the batch, the consumer might simply abort the batch altogether and unsubscribe from the topic, then resubscribe again. This will force a group membership change (it will cause two changes, to be precise) that will see the consumer receive a new partition assignment and start again.

However logical it might seem, this approach does not work. If the consumer is faced with a batch

> where the first, or indeed all records appear to be processed, then bailing out and resubscribing to the topic *might* advance the consumer's offsets to skip over the processed records, but it equally might not. This depends on whether the original consumer had committed its offsets in the first place. Consider a simple scenario involving a single partition with a number of records, which were previously processed by some consumer. The consumer crashed after processing the records but before committing its offsets. A new consumer takes over the topic, and receives the offset of the first record. It detects a duplicate and attempts to resubscribe, but the new assignment will give it the same starting offset — leading to a perpetual backoff cycle on the consumer. Having detected duplicate handling of a record, the correct procedure is to move on to the next record, doing so until the backlog has been worked through.

As a strategy, external fencing may be used in a broad range of scenarios as it does not mutate the topic or sacrifice any notion of safety. Other than the added complexity of requiring an external DLM, the drawback of fencing is its negative impact on throughput and latency. Since record processing must be surrounded by network I/O and persistent operations on the DLM, fencing may pose a challenge for both latency-sensitive and throughput-sensitive applications. This can be alleviated by reorganising records returned from `poll()` into disjoint ordered sets of key-related records, then processing these sets with dedicated locks. In the simplest case, the disjoint ordered sets may be derived from the partition number of the records — each record is allocated to a set based on its partition number. The sets are then processed either sequentially or concurrently on the consumer. Before processing each set, the consumer acquires a lock covering all elements in the set, releasing the lock upon the completion of the set. By coalescing visits to the critical section, the consumer reduces both the network I/O and the load on the DLM. This optimisation is illustrated in the sequence diagram below.



Critical region — disjoint sets

When designing a distributed consumer ecosystem, care must be taken to ensure that the DLM does not become a single point of failure. While focusing on safety, one must take care to not neglect liveness; if the DLM becomes unavailable, the entire consumer ecosystem will stall.

One does not have to use a dedicated DLM for arbitrating access to critical sections. Some persistent datastores, such as relational databases, Etcd, Redis, and Consul can act as locking primitives. (Any store that is both persistent and exposes a compare-and-swap operation may be used to construct a mutex.) In the case of a relational database (and some NoSQL systems), the database can be used to affect transactional semantics over a series of operations — using pessimistic locking or multi-version concurrency control. When the critical section is bound to a single database, the transactional capabilities of the database should be preferred over a DLM, as this minimises the amount of network I/O, simplifies the design of the application, and leads to a stronger consistency model.

> Notwithstanding the various controls one might employ to ensure partition exclusivity, it can be shown that in the absence of a deterministic system, the safety property cannot be unconditionally satisfied in all cases.
>
> Sources of nondeterminism in the execution of applications, such as interrupts, paging, heap compaction, garbage collection, and so forth, can lead to undetectable gaps in the evaluation of successive statements. Consider the following code, typical of 'safe' access to a critical section.

```
distributedLock.acquire();
try {
  criticalSection.enter();
  // ...
  criticalSection.leave();
} finally {
  distributedLock.release();
}
```

> The acquisition of an entry permit to a critical section may be invalidated before the section is subsequently entered due to an event that cannot be foreseen or corrected, for example, a garbage collector pause. In the example above, the `criticalSection.enter()` call may be preceded by an abnormally long pause, such that the DLM may deem the lock holder 'dead' and transfer the lock to the next contender. When the code eventually resumes executing and calls `criticalSection.enter()`, it will conflict with the new lock holder's actions and violate the safety property.

The third strategy — process-level fencing — augments strategy #2 by identifying and restarting (or otherwise isolating) processes that have had their partitions revoked as a result of a heartbeat failure. This can be accomplished using an external health check mechanism that continually queries the process on a well-known endpoint — to determine whether the process can satisfy its routine polling obligations. If the endpoint does not respond within a set timeframe, or if the response is negative, the process will be forcibly restarted. The health check should be tuned to respond negatively well before the `max.poll.interval.ms` deadline elapses, so that the process can be restarted before its death is detected on the coordinator and well before its partitions are reassigned.

Utilising static members

As mentioned earlier in this chapter, Kafka offers the notion of static group membership — allowing for an external health check and healing mechanism for ensuring the liveness of the consumer ecosystem. The requirement for static membership stems from the more contemporary use cases involving container orchestration engines such as Kubernetes. Under the 'Kubernetes' model, the container orchestrator is responsible for the ongoing health check of its subordinate containers, restarting them if a failure is detected.

Kubernetes, due to its overwhelming popularity, is often quoted as a ubiquitous 'placeholder' term for any control system that takes on additional health assurance responsibilities. When 'Kubernetes' is written in quotes, as in the preceding instance, one should assume that the conversation relates to any control system.

In the 'Kubernetes' deployment model, a container may be terminated spuriously as a result of a failed health check, only to be restarted shortly afterwards. It would be highly advantageous for the partitions assigned to the failed consumer to remain as such, with no intermediate reassignment, until the consumer rejoins the group. This model avoids a stop-the-world pause at the expense of individual partition availability — one or more partitions (depending on the failed consumer's assignment) will remain paused for some time. This may result in greater overall throughput and more pronounced positive skew in the distribution of response times — lower median latencies at the expense of a longer latency tail.

Under the static model, members are allowed to leave and join a group without forfeiting their partition assignment or causing a rebalance, providing that they are not away for longer than the `session.timeout.ms` deadline; failing to reappear within the heartbeat deadline causes a purge, triggering a subsequent rebalance. In this respect, there is no material difference in the liveness behaviour between the static and dynamic protocols from the coordinator's perspective.

Naturally, if one is using an external orchestrator for managing consumer health, it makes sense to desensitise the coordinator's own health check mechanism to prevent a premature expulsion of the bounced consumer, but still allow the coordinator act as a 'safety net' if the orchestrator does not detect and remediate the process failure in a timely manner. This can be achieved by setting the `session.timeout.ms` deadline to a value that exceeds the orchestrator's projected response time by some margin. The `max.poll.interval.ms` deadline would also be elevated accordingly.

The effect of `max.poll.interval.ms` is different under the static group membership scheme. The `KafkaConsumer` client will cease to heartbeat if the deadline is not met by the poll-process loop, but no further action will be taken — namely, the client will not send a leave request and no reassignment will occur. The stalled consumer process will be given up to `session.timeout.ms` to recover. If no heartbeat is received within the `session.timeout.ms` deadline, only then will the coordinator evict the failed consumer and reassign its partitions. In the course of introducing the static membership feature, the hard upper bound on `session.timeout.ms` was increased to `1800000` (30 minutes).

With the additional consumer health management options offered by static group membership, one might wonder whether this changes the liveness and safety landscape. The somewhat grim answer is: not by a lot.

The liveness property is fundamentally satisfied by identifying consumer failures. Whether the failure is identified by the coordinator or by an external process is mostly an implementation detail. A static model may reduce the impact of intermittent failure on the group, at the expense of individual partition stalls.

The safety property is achieved by conveying an exclusive partition assignment, on the assumption that the assignment is commensurately honoured on the consumers. We previously looked at process fencing as a mechanism for ensuring partition exclusivity among consumers. Static membership, coupled with an external orchestrator, offers a straightforward implementation path for the fencing strategy. The process health check will factor into account the last poll time — reporting a heavily backlogged process as failed. The orchestrator will restart the consumer process, causing it to rejoin with its durable group instance ID, restoring any prior assignments. The consumer's progress in the stream will be reset per the last committed offsets, allowing it to reprocess the last batch. The combination of locking critical sections and fencing processes effectively neutralises any residual likelihood of a lingering 'zombie' process.

Mixing membership models

The static membership model on its own is generally incompatible with the notion of consumer elasticity. With a programmatic assignment of the `group.instance.id` property, it is possible to accommodate the scale-up scenario. A newly spawned static consumer joining a group would acquire a share of partitions, working in much the same way as a dynamic consumer. The problem occurs on the scale-down path: a terminated static consumer will lead to partition unavailability for the period of `session.timeout.ms`, which has presumably been increased as part of switching to static membership. In other words, scale-up will be responsive while scale-down will be sluggish.

It is possible to mix elastic and inelastic consumers in the same consumer group. Consider a use case where an application is experiencing a mostly constant load throughout the day. Occasionally, the load peaks above the routine threshold, requiring additional burst capacity to process the stream. Assuming that throughput takes priority over latency and occasional individual partition pauses are acceptable, the consumer population might comprise a mixture of statically-configured and dynamic consumers. When utilising public cloud infrastructure, it is more cost-effective to provision long-term capacity for fixed loads, while utilising ephemeral instances (e.g. Spot Instances in AWS) for spillover capacity. Kafka's static membership feature dovetails nicely into this model.

When mixing consumer types, dynamic consumers should be configured differently for the `session.timeout.ms` property. Dynamic consumers will have a much shorter `session.timeout.ms` deadline, letting Kafka manage consumer liveness. Static consumers will have a more relaxed `session.timeout.ms` deadline, but will also typically utilise another orchestrator with a more aggressive health check interval.

Partition assignment strategy

All considerations around group membership ultimately serve one purpose — to ensure that partitions are assigned among the members of a consumer group. As previously discussed, partition assignment takes place on a single consumer — the leader of the group.

The principal motivation for performing assignment on a consumer process is that it enables the application developers to substitute one of the built-in assignment strategies with a custom one,

without having to include the implementation of the custom strategy in the classpath of each of the broker nodes. This is particularly crucial when a cluster is being utilised in a multitenancy arrangement, where different teams or even paying customers might share the same set of brokers. It would be intractable to redeploy a broker upon a change to the rebalancing strategy, due to the disruption this would cause to all clients. Under the consumer-led model, one can easily roll out an update to the assignment strategy by redeploying the consumers, with no change or interruption to the Kafka cluster.

One of the challenges of the consumer-led model is the possible disagreement as to the preferred assignor implementation among the population of consumers. The leader will attempt to determine the most-agreed-upon strategy from the list of preferred strategies submitted by each member. If no common strategy can be agreed upon, the leader will return an error to the coordinator, which will be propagated to the remaining group members.

A consumer specifies its preferred assignors by supplying a list of their fully-qualified class names via the `partition.assignment.strategy` property. The assignor implementation must conform to the `org.apache.kafka.clients.consumer.ConsumerPartitionAssignor` interface. The assignors appear in the order of priority; the first assignor is given the highest priority, followed by the next assignor, and so on.

Although assignors are specified as class names, they are communicated using their symbolic names in the *state synchronisation* phase of the protocol. This allows for interoperability between different client implementations — for example, Java clients intermixed with Python clients. As long as the assignor names match, the consumers can agree on a common one.

Built-in assignors

There are four built-in assignors at one's disposal. At an overview level, these are:

1. `RangeAssignor`: the default assignor implied in the client configuration. The range assignor works by laying out the partitions for a single subscribed topic in numeric order and the consumers in lexicographic order. The number of partitions is then dividend by the number of consumers to determine the range of partitions that each consumer will receive. Since the two numbers might not divide evenly, the first few consumers may receive an extra partition. This process is repeated for each subscribed topic.
2. `RoundRobinAssignor`: allocates partitions in a round-robin fashion. The round-robin assignor works by laying out all partitions across all subscribed topics in one list, then assigning one partition to the first consumer, the second partition to the next consumer, and so on — until all partitions have been assigned. When the number of consumers is exhausted, the algorithm rotates back to the first consumer.
3. `StickyAssignor`: maintains an evenly-balanced distribution of partitions while preserving the prior assignment as much as possible, thereby minimising the difference in the allocations between successive assignments.
4. `CooperativeStickyAssignor`: a variant of the sticky assignor that utilises the newer cooperative rebalancing protocol to reduce the stop-the-world pause.

Range assignor

The range assignor, along with the round-robin assignor, is among the two of the oldest assignors in Kafka's arsenal. The range assignor is the default, yielding contiguous partition numbers for consumers. (This is easier on the eye, not that it matters much.) To understand how it works, consider the following scenario, where a topic with eight partitions is grown from one consumer to two, then to three, and finally, to four consumers. Initially, consumer assignments would resemble the following:

Consumer	Partitions
C0	P0, P1, P2, P3, P4, P5, P6, P7

Following the addition of a second consumer, the partition assignments would be altered to:

Consumer	Partitions
C0	P0, P1, P2, P3
C1	P4, P5, P6, P7

Comparing the two states, we arrive at the following delta:

Consumer	Assigned	Revoked
C0		P4, P5, P6, P7
C1	P4, P5, P6, P7	

The change encompasses four assignments and four revocations across both consumers — four swaps, so to speak. This is on par with the best assignment schemes, representing the optimal outcome, in terms of minimising reassignments and maintaining an evenly-balanced consumer group. By 'evenly-balanced', it is meant that the difference between the partition count of the largest allocation and that of the smallest allocation cannot exceed one.

Now consider the addition of a third consumer.

Consumer	Partitions
C0	P0, P1, P2
C1	P3, P4, P5
C2	P6, P7

Consumer	Assigned	Revoked
C0		P3
C1	P3	P6, P7
C2	P6, P7	

There are three swaps in this scenario. How would this compare to an optimal assignment? To maintain even balance with minimal reshuffling, we would simply revoke *P3* from *C0* and *P7* from *C1*, transferring them to *C2*:

Consumer	Partitions
C0	P0, P1, P2
C1	P4, P5, P6
C2	P3, P7

Consumer	Assigned	Revoked
C0		P3
C1		P7
C2	P3, P7	

So in this case, the range assignor loses to the optimum by three reassignments to two. While not ideal, this is marginally better than another popular built-in strategy — the round-robin assignor.

Let's increase the consumer count to four.

Consumer	Partitions
C0	P0, P1
C1	P2, P3
C2	P4, P5
C3	P6, P7

Consumer	Assigned	Revoked
C0		P2
C1	P2	P4, P5
C2	P4, P5	P6, P7
C3	P6, P7	

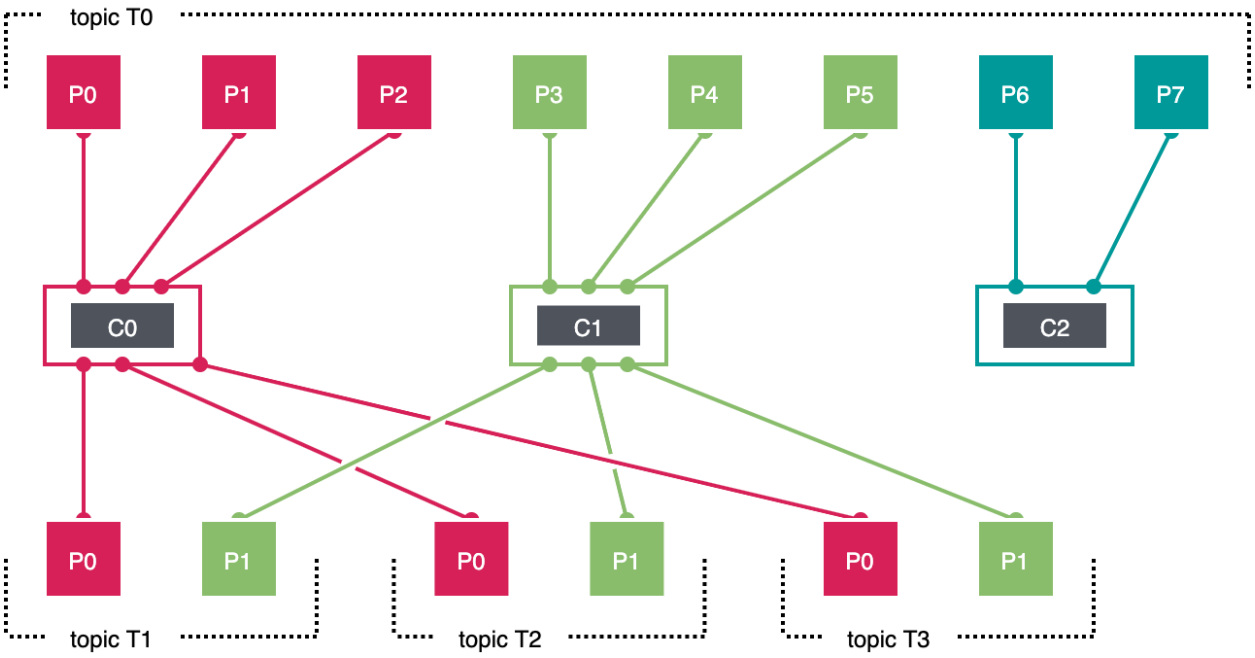
Comparing this with the optimum:

Consumer	Partitions
C0	P0, P1
C1	P3, P4
C2	P6, P7
C3	P2, P5

Consumer	Assigned	Revoked
C0		P2
C1		P6
C2		
C3	P2, P6	

The efficiency of a range assignor drops significantly compared to the optimum — five swaps to two. Beyond three consumers, the preservation of assignments is fairly poor.

Despite being the default option, the range assignor suffers from one severe deficiency: it leads to an uneven assignment of partitions to consumers having multiple topic subscriptions, where the partition count is, on average, smaller than the number of consumers. This phenomenon is illustrated below.



Range partition assignor

The range assignor considers the partitions for each topic separately, allocating them evenly among the consumers. In the above example, the first two consumers get an extra partition each for topic *T0*, which is reasonable, since 8 does not divide evenly into 3. So far, the range assignor seems fair.

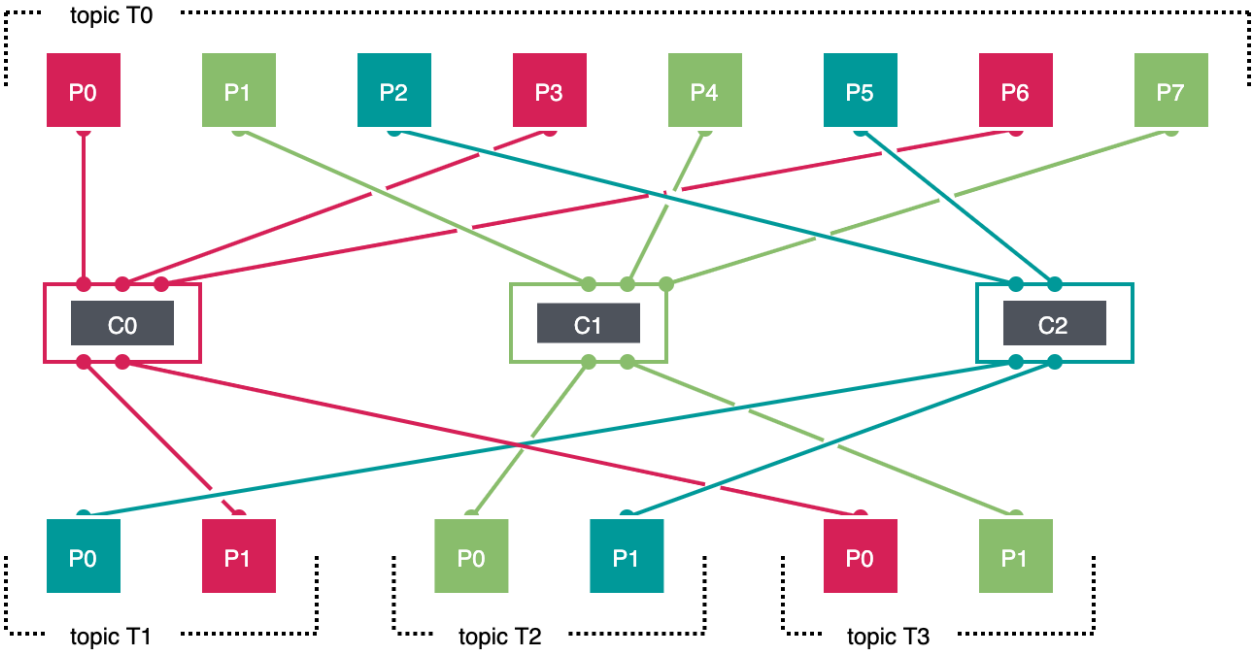
Moving on to *T1*, the two available partitions are assigned to consumers *C0* and *C1*, respectively. With insufficient partitions in *T1*, *C2* is left without an assignment. Again, this does not appear unreasonable when taken in isolation. However, when considering that the assignment for *T0* also favoured the first two consumers, it seems that *C2* is at a slight disadvantage, being towards the end of the pack. The inherent bias becomes more pronounced as we consider *T2* and *T3*; there are no new assignments for *C2*.

Kafka's performance doctrine of scaling consumers to increase parallelism does not always apply to its default assignor — under certain circumstances, as evident above, it will not scale no matter how many consumers are added into the mix — the new consumers will remain idle, despite having a sufficient aggregate number of partitions across all subscribed topics. This can be helped by ensuring that each topic is sufficiently 'wide' on its own. However, the range assignor is still inherently biased when the number of partitions does not divide cleanly into the number of consumers — giving preference to the consumers appearing higher in the lexicographical order of their member ID.

Round-robin assignor

Unlike the range assignor, the round-robin strategy aggregates all available partitions from every subscribed topic into a single vector, each element being a topic-partition tuple, then maps those topic-partitions onto the available consumers. In doing so, the round-robin assignor creates an even loading of consumers, providing that the subscription is uniform — in other words, each consumer is

subscribed to the same set of topics as every other consumer. This is usually the case in practice; it is rare to see a heterogeneous set of consumers with different subscription interests in the same group. The diagram below illustrates the effectiveness of a round-robin strategy in achieving a balanced loading.



Round-robin partition assignor

The preservation properties of a round-robin assignor are typically only slightly worse than that of the range assignor. Consider the previous test of subjecting a topic with eight partitions to a growing consumer population. Starting with the one-to-two (consumers) transition:

Consumer	Partitions
C0	P0, P2, P4, P6
C1	P1, P3, P5, P7

Consumer	Assigned	Revoked
C0		P1, P3, P5, P7
C1	P1, P3, P5, P7	

The performance is on par with the optimum. In fact, it can be trivially shown that every strategy will perform identically to the optimum for the one-to-two transition, in either direction.

Increasing the consumer population to three:

Consumer	Partitions
C0	P0, P3, P6
C1	P1, P4, P7
C2	P2, P5

Consumer	Assigned	Revoked
C0	P3	P2, P4
C1	P4	P3, P5
C2	P2, P5	

There are four swaps now, whereas two would have been optimum.

Now for the three-to-four transition:

Consumer	Partitions
C0	P0, P4
C1	P1, P5
C2	P2, P6
C3	P3, P7

Consumer	Assigned	Revoked
C0	P4	P3, P6
C1	P5	P4, P7
C2	P6	P5
C3	P3, P7	

The performance degrades to the same level as the range assignor, in both cases taking five swaps compared to the two swaps of the optimum strategy.

All in all, the round-robin assignor will show marginally worse preservation qualities than the range assignor in some cases. But in all cases, the round-robin assignor provides an ideal distribution of load in a homogeneous subscription. Frankly, one struggles to comprehend why the range assignor was chosen as the default strategy over round-robin, given that both are of the same vintage and the latter is more suitable for a broader range of scenarios.

Furthermore, the cost of rebalancing partitions is mostly attributable to the consumer ecosystem; consumers will contribute to the stop-the-world pause by blocking in their `onPartitionsRevoked()` callback. (Even without the optional callback, there will be some amount of cleanup within the `KafkaConsumer`, such as committing offsets if offset auto-commit is enabled.) Since both assignors utilise the eager rebalancing protocol, both sets of consumers will equally assume the worst-case scenario and revoke all partitions before synchronising state.

Sticky assignor

The observable degradation in preservation abilities of range and round-robin assignors as the number of consumers increases is attributable to their statelessness. The range and round-robin assignors arrive at the new allocations by only considering the requested subscriptions and the available topic-partitions. The sticky assignment strategy improves upon this by adding a third factor — the previous assignments — to arrive at an optimal number of swaps while maintaining an ideally balanced group. In other words, the sticky assignor performs as well as the theoretically-optimum strategy and retains the balancing qualities of round-robin.

The main issue with using a sticky assignor is that it is still based on the aging eager rebalance protocol. Without special consideration of the assignment strategy in the rebalance listener callback, a consumer application will still be forced down a worst-case assumption concerning partition revocations.

The documentation of the `StickyAssignor` suggests the following approach to mitigate the effects of the eager protocol. Ordinarily, a `ConsumerRebalanceListener` will commit offsets and perform any additional cleanup in the `onPartitionsRevoked()` callback, then initialise the new state as so:

```
new ConsumerRebalanceListener() {
  void onPartitionsRevoked(Collection<TopicPartition> partitions) {
    for (var partition : partitions) {
      commitOffsets(partition);
      cleanupState(partition);
    }
  }

  void onPartitionsAssigned(Collection<TopicPartition> partitions) {
    for (var partition : partitions) {
      initState(partition);
      initOffset(partition);
    }
  }
}
```

Under a sticky assignment model, one may want to defer the `cleanupState()` operations until after the partitions have been reassigned, providing that it is safe to do so. The amended listener code:

```
new ConsumerRebalanceListener() {
  Collection<TopicPartition> lastAssignment = List.of();

  void onPartitionsRevoked(Collection<TopicPartition> partitions) {
    for (var partition : partitions)
      commitOffsets(partition);
  }

  void onPartitionsAssigned(Collection<TopicPartition> partitions) {
    for (var partition : difference(lastAssignment, assignment))
      cleanupState(partition);

    for (var partition : difference(assignment, lastAssignment))
      initState(partition);
```

```
    for (var partition : assignment)
      initOffset(partition);

    this.lastAssignment = assignment;
  }
}
```

As a matter of fact, this approach could be used with any assignor; however, the sticky assignor ostensibly makes this more worthwhile. Whether this holds in practice, and to what extent, largely depends on the cost of the `cleanupState()` and `initState()` methods, especially if the former needs to block.

If `cleanupState()` is a blocking operation, then at least it will not hold back the `onPartitionsRevoked()` method when deferred, meaning that it will not contribute to the stop-the-world pause. By the same token, any consumer can alter its `ConsumerRebalanceListener` callbacks to take advantage of a deferred cleanup, irrespective of the chosen assignor. In other words, partition stickiness on its own does not yield an advantage in `onPartitionsRevoked()` — the gains result from refactoring and are independent of the chosen strategy. The sticky assignor helps in reducing the number of `cleanupState()` and `initState()` calls that are outside of the stop-the-world phase. The latter is significant, as it delays the start of record processing on the consumer, but does not affect the other consumers. The `cleanupState()` method, on the other hand, should ideally be executed asynchronously if possible — it is hard to imagine why state cleanup should hold up the `onPartitionsAssigned()` callback.

Cooperative sticky assignor

The cooperative sticky assignor is a relatively recent addition to the family, appearing in Kafka 2.4.0. It produces the same result as its eager `StickyAssignor` predecessor, utilising the newer incremental cooperative rebalance protocol.

The main difference between the two variants is that in the cooperative case, the consumer is told exactly which partitions will be revoked, reducing the blocking time of the `onPartitionsRevoked()` callback and alleviating the stop-the-world pause.

Upgrading assignors and rebalance protocols

As the assignor configuration is defined client-side, one may be tempted to change the assignor by simply updating the `partition.assignment.strategy` property to a new assignor and then bouncing the consumers. In reality, doing so in the presence of multiple consumers will halt the consumer group. When the first client bounces and rejoins the group, it will mandate an assignor that isn't supported by the remaining members, leading to an `INCONSISTENT_GROUP_PROTOCOL` error emitted by the group leader, manifesting as an `InconsistentGroupProtocolException` on the client.

The correct mechanism for changing an assignor is to perform two bounces. Before the first round, change the consumer configuration to reflect both the preferred and the outgoing assignor.

For example, if changing from the default range assignor to the cooperative sticky assignor, the `partition.assignment.strategy` property should be set to `org.apache.kafka.clients.consumer.RangeAssignor,c`. Having completed the first round, update the configuration to exclude the outgoing assignor. The assignors may initially appear in either order; the priority difference will be resolved by the start of the second round. After two rounds, the client configuration should only contain the cooperative assignor. This process ensures that at every point there is at least one common protocol that every consumer is willing to accept.

When migrating from the default consumer, it is essential that the `org.apache.kafka.clients.consumer.RangeAssig` is included explicitly in the strategy list, as the client will not add it by default.

This chapter has taken the reader on a deep-dive tour of Kafka's elaborate group management protocol — the underlying mechanism for dealing with consumers 'coming and going' gracefully. And not too gracefully, as it has turned out — consumers may spuriously disappear due to intermittent failures or reappear unexpectedly as lingering 'zombie' processes.

The ultimate intent of group membership is to allocate partitions to consumers and ensure exclusivity. As we have learned, the mechanics of partition assignment are fulfilled by an embedded protocol — an encapsulated communication between an elected group leader and the subordinate members, arbitrated by a dedicated coordinator node. The actual assignment is courtesy of a `ConsumerPartitionAssignor` implementation that executes entirely on the lead consumer; several built-in assignment strategies are available, catering to the needs of most consumers.

More recently, Kafka has improved support for container orchestration technologies like Kubernetes, improving rebalance performance and opening doors to external mechanisms for managing consumer health — means for promptly identifying and swiftly dealing with failures.

If there is anything that this chapter has imparted, it is that Kafka is no trivial creation. It is a complex beast, designed to fulfill a broad range of stream processing needs. For all its flexibility and configuration options, employing Kafka correctly is not straightforward by any stretch of the imagination. One requires a firm grasp on the theory of distributed and concurrent computing — at minimum, a thorough appreciation of the *liveness* and *safety* properties. *Anything that can go wrong, will go wrong* — as it was once famously said. Hopefully, the material in this chapter has prepared the reader for all sorts of situations.

Chapter 16: Security

With the phenomenal level of interconnectedness prevalent in the modern world, opportunities arise not just in legitimate commercial enterprises, but also in the more clandestine establishments — individuals or organisations that seek to capitalise on unprotected systems and networks, with an overarching motive to threaten their targets for the purpose of extortion, cause immediate and lasting harm, or directly profit from illicit activities. The topic of conversation is, of course, cybercrime and information security.

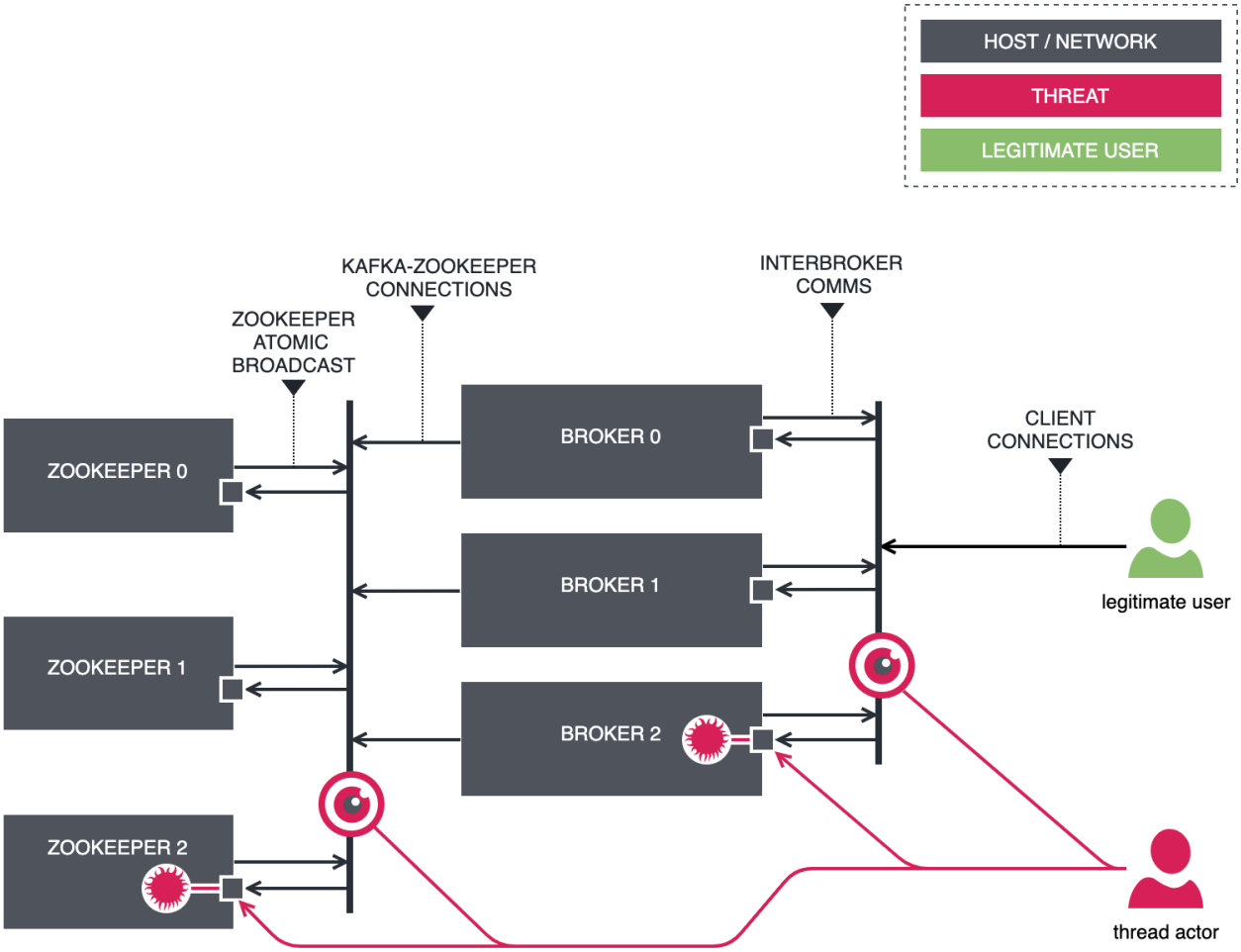
There is a common misconception that cybercriminals target an organisation's most secure defences and employ highly sophisticated techniques to penetrate defences. This view has been largely popularised in modern fiction, where the persona of a hacker has been romanticised as a lone, reclusive individual with above-average skills who acts out of spite or takes their target as a personal challenge. In practice, it is more instructive to view cybercrime as a business, albeit an illegitimate one. While just about any target is theoretically 'hackable', some are just not worth it, insofar as the cost of breaking into an organisation may outweigh the prospective gains. Like any business, a criminal organisation will seek to profit with minimal outlay, which means going after the easy prospects, with trivial or non-existent defences. And where an organisation is heavily defended, cybercriminals will often probe for alternate ways in, looking for weaknesses at the perimeter, as well as from inside the organisation — using social hacking to gain access. As such, it is imperative to utilise multi-layered defences, creating redundancy in the event a security control fails or a vulnerability is exposed

Kafka is a persistent data store that potentially contains sensitive organisational data. It often acts as a communication medium, connecting disparate systems and ensuring seamless information flow within the organisation. These characteristics make Kafka a lucrative target to cybercriminals: if breached, the entire organisation could be brought to its knees. As such, it is essential that particular attention is paid to the security of Kafka deployments, particularly if the cluster is being hosted among other infrastructure components or if it is accessible from outside the perimeter network.

State of security in Kafka

Let us start by making one thing abundantly clear: *Kafka is not secure by default*. Kafka has numerous security controls that cover virtually all aspects of information security; however, these controls are disabled out of the box.

The default Kafka security profile is illustrated below.



Out-of-the-box state of Kafka security

Specifically, the notable areas of shortfall are:

- **Any client can establish a connection to a ZooKeeper or Kafka node.** This statement doesn't just cover the conventional ports `2181` (ZooKeeper) and `9092` (Kafka) — diagnostic ports such as those used by JMX (Java Management Extensions) are also available for anyone to connect to. What makes this setup particularly troubling is that a connection to ZooKeeper is not essential for the correct operation of legitimate clients. While earlier versions of Kafka required clients to interface directly with ZooKeeper, this requirement has been removed as of version 0.9 — a release that dates back to November 2015. ZooKeeper is a highly prized trophy for potential attackers, as writing to *znodes* (internal data structure used to persist state within ZooKeeper) will trivially compromise the entire cluster.
- **Connections to Kafka brokers are unencrypted.** Connections are established over TCP, with the parties exchanging data over a well-documented binary protocol. A third party with access to the transit network may use a basic packet sniffer to capture and analyse the network traffic, obtaining unauthorized access to information. A malicious actor may alter the information in

transit or impersonate either the broker or the client. In addition to client traffic, interbroker traffic is also insecure by default.

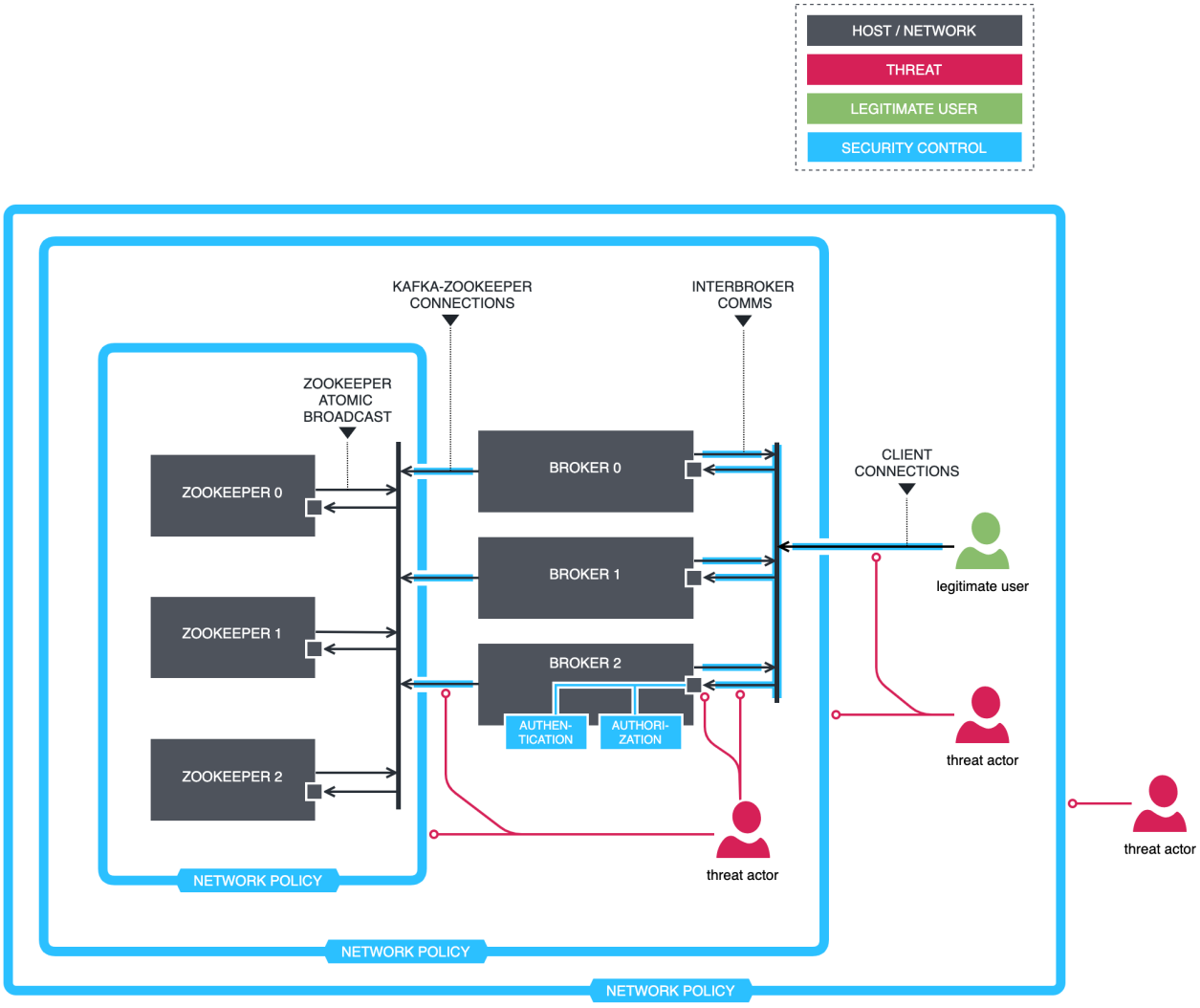
- **Connections to Kafka brokers are unauthenticated**. Any client with knowledge of the bootstrap endpoints can establish a connection to the brokers, appearing as a legitimate client to Kafka. A client does not have to identify itself, nor prove that the supplied identity is genuine.
- **No authorization controls are in place**. Even if authentication is enabled, a client may perform any action on a broker. The default authorization policy is *allow-all*, letting a rogue client run amok.

The reason for Kafka's relaxed default security profile is largely historical. Early versions of Kafka had no support for security, and introducing mandatory security controls would have impacted compatibility with existing clients.

Target state security

Before embarking on the journey of progressively hardening the cluster, it is worthwhile to model an ideal target state. This will act as a notional reference, allowing us to chart a course and ensure that we stick to it.

An overview of the target state is illustrated below. A further examination of the various security principles follows.



Target state security

Minimise the attack surface

There are several ways clients can connect to the Kafka and ZooKeeper cluster, and most of these ingress points are of no use to legitimate clients. On the flip side, they create a large attack surface area, making it difficult to enforce. As such, we need to limit access to the cluster at the lowest possible level, exposing the minimally-essential set of endpoints for legitimate clients.

In addition to limiting ingress, we may also want to explicitly allow specific groups of clients. We might only allow clients from the organisation's internal network to access the cluster. But we may go further than that, segregating traffic based on the clients' location within the internal network. For example, we may partition the internal network into multiple segments, and only allow clients deployed in the operational network (such as the data centre) to access Kafka, while blocking clients in the general-purpose (office) network.

There may also be legitimate reasons to allow external clients to connect; for example, if those clients reside on a remote network that is also trusted, but the two networks are separated by an untrusted network, such as the public Internet. Often this scenario occurs in a hybrid deployment topology, where some parts of an organisation's IT systems may be deployed in one or more data centres, while other parts may be deployed in the public Cloud, or multiple Clouds in different geographical regions, for that matter.

Ensure traffic confidentiality

Traffic flowing in and out of the cluster, and within the cluster, needs to be protected from eavesdropping and tampering. Ordinarily, this implies some sort of encryption, a message signing protocol, and a secure key exchange protocol that collectively assure the end-to-end confidentiality of messages.

Encryption protocols must be chosen such that they comply with industry-accepted standards for the secure exchange of information. These standards exist for a broad set of applications, curated by organisations such as the National Institute of Standards and Technology (NIST). Specifically, NIST breaks standards down into cryptographic hash functions, symmetric key algorithms, and asymmetric key algorithms — all of which will apply to Kafka at some point during the connection lifecycle. In addition to general-purpose standards, specific industry bodies — such as PCI, as well as legislative acts — such as HIPPA, GLBA and SOX, will mandate security controls in addition to the baseline. When deploying Kafka for applications that are expected to comply with regulation, care must be taken to ensure that the applications' approach to information security is equally compliant.

Know the client

Kafka clients are typically embedded into or act on behalf of other applications that play a role in the overall architecture landscape. Identifying clients is a prerequisite for access control; if we don't know who our clients are, we cannot possibly guarantee that access has been restricted to the minimum set of actions that a client is reasonably expected to undertake. A reader familiar with the principles of information security will recognise this statement as the *Principle of Least Privilege* (PoLP). To be specific, attesting the identity of a client does not in itself realise PoLP, but is nonetheless essential for subsequent controls whose role it is to categorically enforce this principle.

Limit access to essential data and functionality

This is the direct enforcement of PoLP. We need to identify all prospective principals — users of the cluster — and determine which resources in the cluster they are allowed to operate upon. By 'resources', it is meant any entity that may pose a material threat if consumed uncontrollably. These include topics, consumer groups, brokers, configuration entries, as well as users. The latter is a resource like any other — the uncontrolled modification of users' profiles may lead to a *privilege escalation*, where a less privileged user may gain a level of access above what has officially been allocated to them. It may also lead to a *denial of service*, where a legitimate user has been blocked from carrying out their normal functions.

With an aspirational target state defined, the scene is set — the rest of the chapter is focused on fulfilling these aspirations — getting Kafka to a state of acceptable security.

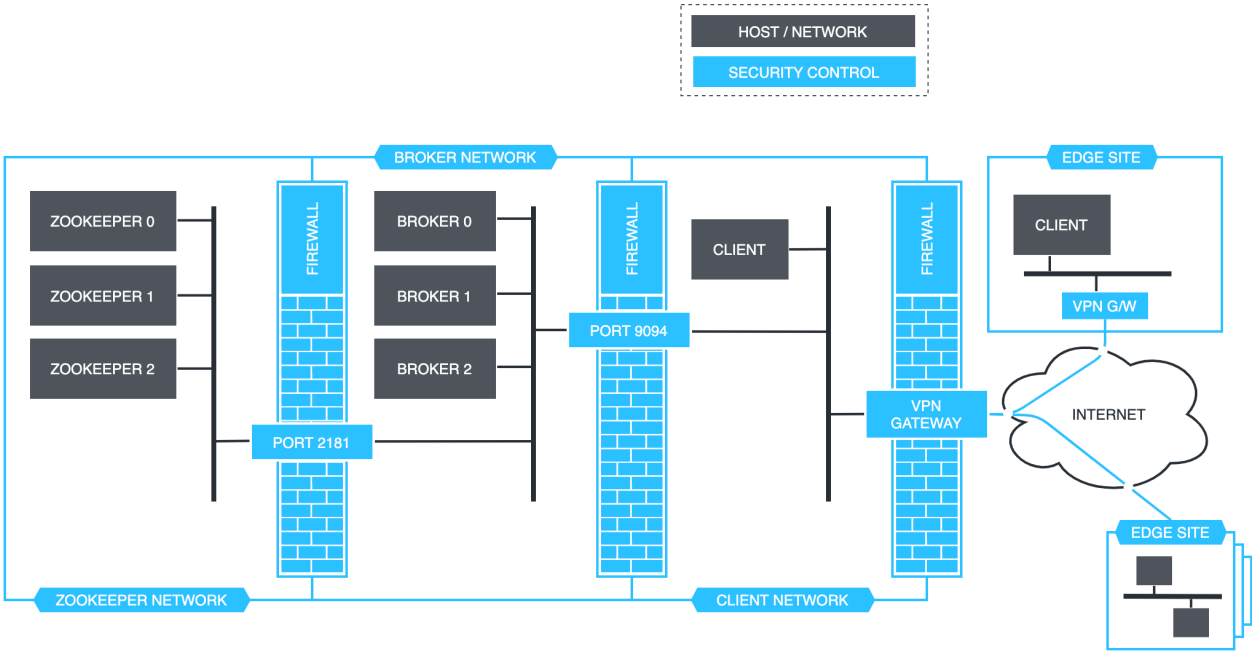
Network traffic policy

Speaking to the earlier point on reducing the attack surface, no control is more effective than a network-level traffic policy — or as it is more commonly known, a firewall.

> Without burrowing into the details, the defining role of a firewall is to segregate traffic between two network segments, such that only legitimate traffic is permitted to traverse the firewall. Conversely, illegitimate traffic (which may be malicious or accidental) is not permitted to pass through the firewall. Network and transport-level controls tend to be simple, straightforward rules that restrict the flow of packets. Because of their simplicity, they are both efficient and difficult to bypass. Especially when used in conjunction with a default-deny policy, one can confidently reason about the behaviour of a firewall, and more importantly, verify its behaviour using real and simulated traffic.

This chapter is not going to examine the different firewall types and how one would practically configure and maintain a firewall. The technology options and physical deployment topologies are just too varied. The physical architecture will further be complicated with the introduction of geographical regions or the use of multiple redundant hosting locations. Rather than attempting to prescribe a physical topology, the intention is to capture the network architecture at a logical level.

At a minimum, the following network partitioning scheme is recommended.



Network segregation

The reason that the firewall port on the broker network is listed as 9094 rather than 9092 is to accommodate an alternate listener, blocking the default one. More on that later.

In the arrangement above, the network is partitioned into four static segments. The ZooKeeper ensemble operates within its own, heavily isolated network segment. This might be a subnet or a series of mutually-routable subnets. The latter would be used in a multi-site deployment, where disparate physical networks are used to collectively host the ensemble. One may also use an overlay network, or other forms of virtual networks, to merge physically separate networks into a single logical network. Again, we must abstain from the detail at this point, and stick to the logical viewpoint.

The brokers operate within a dedicated network segment, which is separated from the ZooKeeper segment by way of a stateful firewall. This ensures that legitimate connections can be established from the brokers to the ZooKeeper nodes, but only those connections and nothing else. No other network segment is permitted to access the ZooKeeper segment. In practice, there may be a need to make specific exclusions for administrative purposes, so that authorized operators of Kafka are able to make changes to the underlying ZooKeeper configuration, deploy software updates, and so forth.

The next outermost segment is the internal client network. This includes producer, consumer and admin clients — essentially the full suite of internal applications that rely on Kafka to shuttle event data. There may be multiple such client networks, or the client network may be fragmented internally. This often happens in hybrid deployment topologies involving different sites. But the topology is more or less stable; in other words, we don't expect entire sites to come and go on a whim.

The fourth network segment is everything outside the perimeter, which can simply be treated as the external network. In the simplest case, the external network is the public Internet. However, this is a relative term. In more elaborate segregation topologies, the external segment might still be within the organisation's perimeter, but outside the client network. It may be the general-purpose staff network or some other corporate intranet. And while in a relative sense, this network might be more trusted than the public Internet, its level of assurance is still largely classified as a 'walk-in' network — it should not be allowed to interface directly with the key technology infrastructure underpinning the revenue-generating systems that propel the organisation.

In addition to the static segments, there may be a case for one or more dynamic network segments for the client ecosystem that could appear and disappear at short notice. A common use case is *edge computing*, where sites may be spun up 'closer to the action', so to speak, but the sites themselves might not be long-lived or may change their location. There may be other ways to allow remotely-deployed applications to utilise Kafka — for example, via dedicated APIs. However, Kafka does not preclude remote access and in some cases this may be desirable — for example, the remote site may have a requirement to persist events internally but might not have a local deployment of Kafka at its convenience. Edge computing dovetails nicely into Kafka's authorization controls, which will be discussed later.

Perhaps the most common use case for remote connectivity is telecommuting. Engineers may be working from a variety of locations, which could be a mixture of private networks, public network, fixed location and mobile networks. Clearly, these sites cannot be easily controlled and are therefore labelled as untrusted. Nonetheless, telecommuters will require first-class access to backend systems and key technology infrastructure to maintain productivity. Although some remote sites may be untrusted, the individual hosts may still be trusted — for example, a remote worker accessing the corporate network over a free Wi-Fi in a café.

Edge locations that require direct connectivity into the core client network are best accommodated using secure virtual networks, such as VPNs. Rather than maintaining temporary 'pinhole' firewall rules that allow access from specific locations based on origin network addresses (which may not always be discernible, particularly if the site is behind a NAT device), a VPN can be used to securely span physically separate networks. The sites (or individual hosts) would be dispensed individual credentials, which may be distributed in the form of client-side Digital Certificates. These credentials will be used to authenticate the site (or host) to the central VPN gateway. The VPN gateway typically resides behind the firewall, although some firewall vendors combine VPN and packet-filtering capabilities into a single appliance. (The latter is sometimes called a VPN concentrator.) Kafka need not be aware of edge locations and VPN arrangements. Instead, the responsibility of accommodating remote sites falls upon the client network and its maintainers. As long as remote sites can securely attach to and transit through the client network, they will have access to the Kafka cluster.

Confidentiality

Kafka supports Transport Layer Security (TLS) for encrypting traffic between several key components. Unlike the discussion on network traffic policy, which maintained a purely theoretical stance, this section offers hands-on examples for configuring TLS and securely connecting producer and consumer clients over encrypted links.

> The reader may be familiar with TLS just by virtue of browsing websites. You may have noticed the padlock on the address bar of the browser — this is an indication that the browser is communicating over an encrypted connection. HTTPS is increasingly used in place of HTTP to secure websites; the 'S' in HTTPS stands for 'secure' and implies the use of SSL and TLS algorithms. In nutshell, SSL is obsolete and TLS is the new name of the older SSL protocol. Technically, the term TLS is more accurate, but most people still use SSL. Digital Certificates used to verify the communicating parties are often referred to as 'SSL Certificates', but the name is purely historical — in fact, they can be used to secure both SSL and TLS traffic.

Like most parts of Java and the rest of the world, Kafka uses the term SSL to refer to TLS. The latest version of TLS at the time of writing is TLS v1.3. The latest version supported by Kafka is TLS v1.2. From a security standpoint, the differences between v1.3 and v1.2 are generally considered to be minor — v1.3 removes certain deprecated ciphers that have known exploits, reducing the likelihood of a misconfiguration impacting the security posture. Where v1.3 trumps its predecessor is in the area of performance — specifically, in connection establishment time.

At the time of writing, support for TLS v1.3 is scheduled to be introduced in version 2.5.0 of Kafka. The main reason for the delayed adoption of TLS v1.3 in Kafka is that it introduces a dependency on Java 11, which is a step up from its current reliance on Java 8, but also a break in runtime compatibility. All in all, when correctly configured, TLS v1.2 is considered to be adequate for general use, and mandated by the Payment Card Industry (PCI) Security Standards Council for the Data Security Standard (DSS).

An essential element of TLS is an X.509 Certificate, often referred to as 'Digital Certificate' or simply 'certificate'. At a minimum, a certificate authenticates the server side of a TLS connection to the client, the latter being the entity that initiates the connection, while the former accepts the connection on a TLS socket. In our case, the server is a broker node. By verifying the certificate presented by the broker to the client against either a trusted certificate authority or a pre-agreed self-signed certificate, the client can be assured that the broker is, in fact, who it claims to be. Certificates can also be used in the opposite direction, attesting the identity of a client to the broker. This scenario will be covered later, in the course of authentication.

> In cryptography, X.509 is a standard defining the format of public-key certificates. X.509 certificates are used in TLS/SSL, which is the basis for HTTPS. They are also used in offline applications that

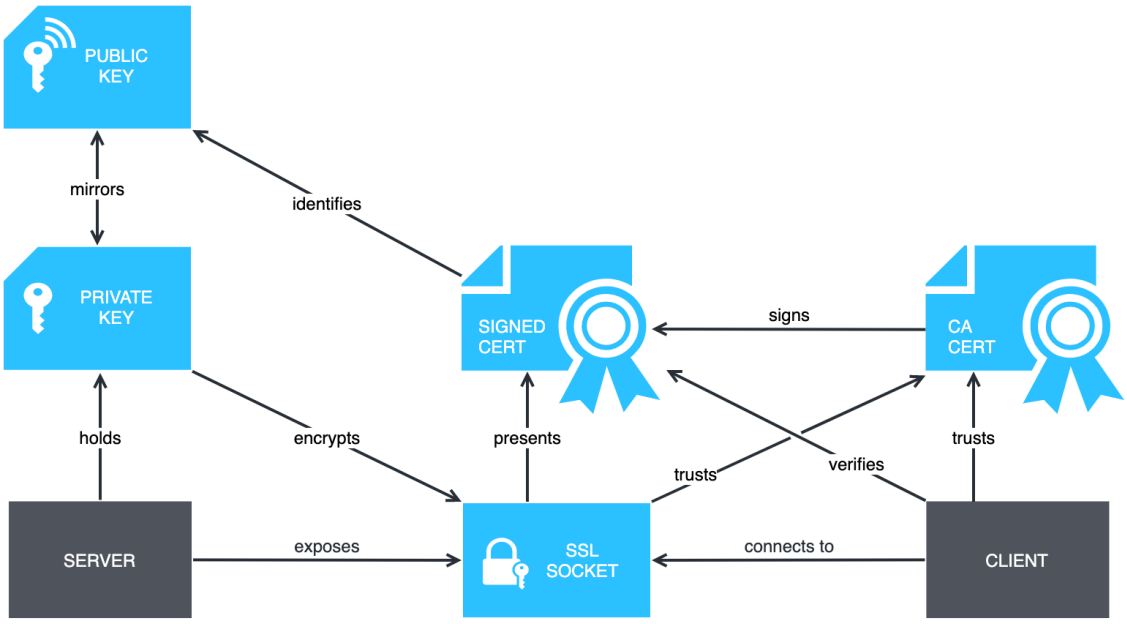
require tamper-proof electronic signatures. An X.509 certificate contains a public key and an identity (a hostname, an organisation, or an individual) — this is depicted in either the Common Name (CN) attribute of the certificate or the Subject Alternative Name (SAN), the latter being an extension to the X.509 base standard.

A certificate is either signed by a *Certificate Authority* (CA) or self-signed. When a certificate is signed by a trusted certificate authority, or validated by other means, someone holding that certificate can rely on the public key it contains to establish secure communications with another party, or validate documents digitally signed by the corresponding private key.

A certificate authority is akin to a government office that issues passports. Passports are printed on specially-crafted materials and use various stamps, watermarks, ultraviolet ink, magnetic strips and holograms to make them difficult to forge. Other governments may inspect the passport using the verification means at their disposal to ascertain its authenticity. Whereas a passport predominantly relies on physical means for ensuring security, the signing of X.509 certificates achieves the equivalent using cryptographic algorithms, with the certificate authority acting as the digital equivalent of a passport office. For as long as the CA remains a genuine and trusted authority, the clients have the assurance that they are connecting to authentic parties. (More recently, passports have also started embedding an RFID chip with a cryptographic module that holds digitally signed biometric data of the passport holder — adding digital security to a traditionally physical device.)

Complementary to X.509 certificates is *Public Key Infrastructure* (PKI) — a comprehensive set of roles, policies, hardware, software and procedures needed to create, manage, distribute, use, store and revoke digital certificates and manage public-key encryption. PKI is an arrangement that binds public keys with respective identities of entities (individuals, hosts and organisations). The binding is established through a process of registration and issuance of certificates at and by a CA. Depending on the assurance level of the binding, this may be carried out by an automated process or under human supervision.

The diagram below illustrates the relationship between a pair of communicating parties — a client and a server — and the supporting mechanisms by which the integrity and confidentiality of the information exchange is guaranteed.



Securing communications using SSL/TLS

It should be stated from the outset that the use of a Certificate Authority and PKI is unequivocally preferred over self-signed certificates. It is not that CA-signed certificates are inherently more secure from a purely cryptographic perspective; rather, the use of PKI makes it easier to manage large numbers of certificates, streamlining the process of issuing, rotating, and revoking certificates. This aspect makes CA-signed certificates more robust overall, as the likelihood of an error is greatly reduced, compared to the bespoke process of managing self-signed certificates. In addition, self-signed certificates must be exchanged out of band, before the parties can communicate securely. This increases the likelihood of them being intercepted and modified en route, which also erodes the trust one may place in self-signed certificates.

For the upcoming examples, we are going to generate our own CA for signing certificates. Naturally, we would have proceeded with a complete PKI scenario, but in practice, the choice of PKI technology elements and their deployment options will be independent to the rest of technology infrastructure choices. Rather than distract the reader with the nuances of PKI, the focus will be on getting a secure connection up and running, with the blissful assumption that PKI is someone else's problem.

To work through the examples below, you will need `openssl` and `keytool` installed. These are open-source packages that will either be pre-installed or can easily be downloaded through a package manager. When generating keys and certificates, you will end up with sensitive material that shouldn't be left unattended, even if it is on your local machine. As such, it is best to create a dedicated directory for all intermediate operations, then delete the directory once you are done. In the upcoming examples, we will be working out of `/tmp/kafka-ssl`.

Java applications use *keystore* and *truststore* files to store keying material. The contents of these files are encoded in a format specific to the Java ecosystem. Keystore files hold private keys and the

associated signed certificates, presented by the client or server applications that hold them. Truststore files house the trusted certificates to authenticate the opposing party. Both files are password-protected. In the case of a keystore, individual keys may be password-protected. The examples in this chapter use the password `secret`. This should be substituted with a more appropriate string.

All examples use a validity period of 365 days for keys and certificates. This period can easily be changed by specifying an alternate value for the `-days` flag (to the `openssl` command) or the `-validity` flag (to `keytool`).

Client-to-broker encryption

We need to generate a key and certificate for each broker in the cluster. The common name of the broker certificate must match the fully qualified domain name (FQDN) of the broker, as the client compares the CN with the resolved hostname to make sure that it is connecting to the intended broker (instead of a malicious one). This process is called *hostname verification*, and is enabled by default. In our examples, we will assume that we are targetting our local test cluster, and therefore the CN will simply be `localhost`. We could use an arbitrary hostname or one containing a wildcard, provided it matches the hostname used in the bootstrap list, as well as the corresponding hostname declared in the `advertised.listeners` broker property.

The client may disable hostname verification, which will allow it to connect to any host, provided that its authenticity can be verified by traversing the certificate chain. However, disabling hostname verification can constitute a serious security flaw, as it makes the client trust *any* certificate that was issued by the CA. This may not be cataclysmic if using a private CA that is constrained to an organisation's PKI; but when a certificate is issued by a public CA, the absence of hostname verification allows any party to impersonate a broker, provided it can hijack the DNS. *Hostname verification is an integral part of server authentication.* Disabling hostname verification may only be safely done in non-production workloads, and even then this practice should be discouraged as it may inadvertently lead to a misconfiguration of production deployments.

Generate the private key

To begin, we need to generate an SSL key and certificate for each broker. We have just one in our test cluster; otherwise, this operation would have to be repeated. (Or better still — automated.) Run the following:

```
keytool -keystore server.keystore.jks -alias localhost \
    -validity 365 -genkey -keyalg RSA
```

This command will prompt you for a password. After entering and confirming the password, the next prompt is for the first and last name. This is actually the common name. Enter `localhost` for our test cluster. (Alternatively, if you are accessing a cluster by a different hostname, enter that name instead.) Leave other fields blank. At the final prompt, hit `y` to confirm.

```
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  localhost
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]:
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]:
Is CN=localhost, OU=Unknown, O=Unknown, L=Unknown, □
    ST=Unknown, C=Unknown correct?
  [no]:  y

Generating 2,048 bit RSA key pair and self-signed certificate □
    (SHA256withRSA) with a validity of 365 days
  for: CN=localhost, OU=Unknown, O=Unknown, L=Unknown, □
      ST=Unknown, C=Unknown
```

The result will be a `server.keystore.jks` file deposited into the current directory.

You can view the contents of the keystore at any time — by running the following command. (It will require a password.)

```
keytool -list -v -keystore server.keystore.jks
```

```
Enter keystore password:
Keystore type: PKCS12
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: localhost
Creation date: 02 Jan. 2020
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=localhost, OU=Unknown, O=Unknown, L=Unknown, □
```

```
    ST=Unknown, C=Unknown
Issuer: CN=localhost, OU=Unknown, O=Unknown, L=Unknown, ␣
    ST=Unknown, C=Unknown
Serial number: 2e0b92ac
Valid from: Thu Jan 02 08:51:53 AEDT 2020 until: ␣
    Fri Jan 01 08:51:53 AEDT 2021
Certificate fingerprints:
   SHA1: 59:2D:AA:C0:A8:B1:CF:B6:F7:CA:B6:C2:21: ␣
      55:44:12:27:44:0F:58
   SHA256: 27:F5:7F:9E:36:A1:B4:0D:72:F6:71:AC: ␣
      A0:8B:F2:BB:06:CA:0C:FD:28:64:86:53:6A:37:BF:EF:81:D0:7F:68
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: B2 00 B4 C1 BA 4A 5E FC   9B 44 B7 29 F3 78 A2 CD
0010: 4A BA 6D 4E
]
]

*******************************************
*******************************************
```

Create a CA

Bearing a private key alone is insufficient, as it does not identify the user of the key or instil trust in it. The following step creates a certificate authority for signing keys.

```
openssl req -new -x509 -keyout ca-key -out ca-cert -days 365
```

You are required to provide a password, which may differ from the password used in the previous step. Leave all fields empty with the exception of the *Common Name*, which should be set to `localhost`.

```
Generating a 2048 bit RSA private key
.........+++
..............+++
writing new private key to 'ca-key'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be
incorporated into your certificate request.
What you are about to enter is what is called a
Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) []:
State or Province Name (full name) []:
Locality Name (eg, city) []:
Organization Name (eg, company) []:
Organizational Unit Name (eg, section) []:
Common Name (eg, fully qualified host name) []:localhost
Email Address []:
```

The above command will output `ca-key` and `ca-cert` files to the current directory.

The next two steps will import the resulting `ca-cert` file to the broker and client truststores. Once imported, the parties will implicitly trust the CA and any certificate signed by the CA.

```
keytool -keystore client.truststore.jks -alias CARoot \
    -import -file ca-cert
```

```
Enter keystore password:
Re-enter new password:
Owner: CN=localhost
Issuer: CN=localhost
Serial number: 8f444e15ad8f7067
Valid from: Thu Jan 02 09:28:45 AEDT 2020 until: □
    Fri Jan 01 09:28:45 AEDT 2021
Certificate fingerprints:
   SHA1: 70:55:42:23:69:A1:EA:E8:13:49:41:CC:C3:CE:□
      A3:7B:CB:25:F8:08
   SHA256: 7E:CC:21:57:5B:8C:FB:90:D9:9E:2B:84:76:□
```

```
      C4:E1:83:D0:2D:B5:D1:17:3A:D2:D5:5A:4D:C5:CB:F3:9B:32:DD
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 1
Trust this certificate? [no]:  y
Certificate was added to keystore
```

Repeat for `server.truststore.jks`:

```
keytool -keystore server.truststore.jks -alias CARoot \
    -import -file ca-cert
```

Generating private keys and the creation of a CA are mutually independent operations and may be performed in either order. The results of these operations will be combined in the signing stage.

Sign the broker certificate

The next step is to generate the certificate signing request on behalf of the broker.

```
keytool -keystore server.keystore.jks -alias localhost \
    -certreq -file cert-req
```

This produces `cert-req`, being the signing request. To sign with the CA, run the following command.

```
openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-req \
    -out cert-signed -days 365 -CAcreateserial


Signature ok
subject=/C=Unknown/ST=Unknown/L=Unknown/O=Unknown/ □
    OU=Unknown/CN=localhost
Getting CA Private Key
Enter pass phrase for ca-key:
```

This results in the `cert-signed` file.

The CA certificate must be imported into the server's keystore under the `CARoot` alias.

```
keytool -keystore server.keystore.jks -alias CARoot \
    -import -file ca-cert


Enter keystore password:
Owner: CN=localhost
Issuer: CN=localhost
Serial number: 8f444e15ad8f7067
Valid from: Thu Jan 02 09:28:45 AEDT 2020 until: □
    Fri Jan 01 09:28:45 AEDT 2021
Certificate fingerprints:
   SHA1: 70:55:42:23:69:A1:EA:E8:13:49:41:CC:C3:CE:A3: □
      7B:CB:25:F8:08
   SHA256: 7E:CC:21:57:5B:8C:FB:90:D9:9E:2B:84:76:C4: □
      E1:83:D0:2D:B5:D1:17:3A:D2:D5:5A:4D:C5:CB:F3:9B:32:DD
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 1
Trust this certificate? [no]:  y
Certificate was added to keystore
```

Then, import the signed certificate into the server's keystore under the `localhost` alias.

```
keytool -keystore server.keystore.jks -alias localhost \
    -import -file cert-signed
```

Resulting in:

```
Enter keystore password:
Certificate reply was installed in keystore
```

Deploy the private key and signed certificate to the broker

With key generation and signing operations completed, the next step is to install the private key and the signed certificate on the broker. Assuming the keystore file is in `/tmp/kafka-ssl`, run the following:

```
cp /tmp/kafka-ssl/server.*.jks $KAFKA_HOME/config
```

Configure the broker to use SSL

Edit `server.properties` and make changes to reflect the following.

```
listeners=PLAINTEXT://:9092,SSL://:9093
advertised.listeners=PLAINTEXT://localhost:9092,SSL://localhost:9093
listener.security.protocol.map=PLAINTEXT:PLAINTEXT,\
    SSL:SSL,SASL_PLAINTEXT:SASL_PLAINTEXT,SASL_SSL:SASL_SSL
inter.broker.listener.name=PLAINTEXT
ssl.keystore.location=\
    /Users/me/opt/kafka_2.13-2.4.0/config/server.keystore.jks
ssl.keystore.password=secret
ssl.key.password=secret
ssl.truststore.location=\
    /Users/me/opt/kafka_2.13-2.4.0/config/server.truststore.jks
ssl.truststore.password=secret
```

Looking over the configuration —

- The first change is the addition of `SSL://:9093` to the `listeners` list. This creates a new server socket, bound to port `9093`.
- The socket is advertised as `SSL://localhost:9093` in the `advertised.listeners`.
- The `listener.security.protocol.map` and `inter.broker.listener.name` properties remain unchanged for this example, as we have not defined a new protocol, nor have we changed how the brokers communicate with each other.
- The addition of `ssl...` properties configures the broker to use the keying material that was generated in the previous steps.

Having saved `server.properties`, restart the broker for the changes to take effect.

In the configuration above, the SSL settings were defined in global scope — applying uniformly to all listeners configured to use TLS, including the interbroker connection (if applicable). Kafka allows us to specify custom SSL settings for individual listeners, by prefixing the `ssl...` property names with the lowercase name of the listener, in the form `listener.name.<lowercase_name>.<setting>=<value>`; for example, `listener.name.external.ssl.keystore.password=secret`.

Deploy the CA certificate to the client

The previous client examples were communicating with the broker over a cleartext connection. In order to enable SSL, we must first copy the `client.truststore.jks` file to our source code directory:

```
cp client.truststore.jks ~/code/effectivekafka
```

Configure the client to use SSL

The complete source code listings for the SSL producer and consumer client examples are available at github.com/ekoutanov/effectivekafka[31] in the `src/main/java/effectivekafka/ssl` directory. The keystore and truststore files have not been committed to the repository, as they vary between deployments.

The following example configures a producer client to use SSL. With the exception of new SSL-related configuration properties, it is otherwise identical to the original producer sample presented in Chapter 5: Getting Started.

```
import static java.lang.System.*;

import java.util.*;

import org.apache.kafka.clients.*;
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.config.*;
import org.apache.kafka.common.serialization.*;

public final class SslProducerSample {
  public static void main(String[] args)
      throws InterruptedException {
    final var topic = "getting-started";

    final Map<String, Object> config = Map
        .of(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "localhost:9093",
            CommonClientConfigs.SECURITY_PROTOCOL_CONFIG,
            "SSL",
            SslConfigs.SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG,
            "https",
            SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,
            "client.truststore.jks",
            SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG,
            "secret",
            ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName(),
            ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName(),
            ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,
```

[31] https://github.com/ekoutanov/effectivekafka/tree/master/src/main/java/effectivekafka/ssl

```
            true);

    try (var producer = new KafkaProducer<String, String>(config)) {
      while (true) {
        final var key = "myKey";
        final var value = new Date().toString();
        out.format("Publishing record with value %s%n",
                   value);

        final Callback callback = (metadata, exception) -> {
          out.format("Published with metadata: %s, error: %s%n",
                     metadata, exception);
        };

        // publish the record, handling the metadata in the callback
        producer.send(new ProducerRecord<>(topic, key, value),
                      callback);

        // wait a second before publishing another
        Thread.sleep(1000);
      }
    }
  }
}
```

The first point of difference is the use of the `security.protocol` property, which specifies the use of SSL. This property holds a compound value — not only does it specify the encryption scheme, but also the authentication scheme. As we are not using authentication in this example, its value is simply `SSL`.

The `ssl.endpoint.identification.algorithm` property specifies how the broker's hostname should be validated against its certificate. The default value is `https`, enabling hostname verification. (This is the recommended setting.) To disable hostname verification, set this property to an empty string.

The `ssl.truststore.location` and `ssl.truststore.password` properties are required to access the CA certificate that we have just configured.

The consumer example is analogous and is omitted for brevity.

Interbroker encryption

The changes that were made so far focused on securing the communications between brokers and clients, leaving brokers to mingle using the regular cleartext protocol. Fortunately, configuring Kafka to use SSL for interbroker communications is a trivial matter. Edit `server.properties` and make the following one-line change.

```
inter.broker.listener.name=SSL
```

Restart the broker for the change to take effect. To verify that SSL is being used, check the startup logs. The server should output the current value of the `inter.broker.listener.name` property. You can also run `netstat` to ensure that there are no remaining connections on port `9092`, and that all traffic is now being served on port `9093`.

```
netstat -an | egrep "9092|9093"
```

```
tcp4       0      0  127.0.0.1.9093         127.0.0.1.65134        ESTABLISHED
tcp4       0      0  127.0.0.1.65134        127.0.0.1.9093         ESTABLISHED
tcp46      0      0  *.9093                 *.*                    LISTEN
tcp46      0      0  *.9092                 *.*                    LISTEN
```

The filtered output of `netstat` shows that, although both ports are listening, only `9093` is fielding an active connection.

Once it has been verified that SSL is working as planned, both for the client-broker and interbroker use cases, the recommended next step is to *disable the cleartext listener*. This is accomplished by updating all brokers and all clients first, allowing for both cleartext and SSL connections in the interim. Only once all clients and brokers have switched to the `SSL` listener, can the `PLAINTEXT` option be removed.

Disabling the cleartext listener can be done in one of two ways. The most straightforward approach is editing the static configuration in `server.properties` — removing the `PLAINTEXT` entry from both the `listeners` and `advertised.listeners` properties. The second way is to do it remotely, using per-broker or cluster-wide dynamic update modes. Dynamic configuration updates are discussed in Chapter 9: Broker Configuration.

A reader knowledgeable in cryptography may experience a slight confusion around Kafka's somewhat cavalier overloading of the term 'plaintext'. The `PLAINTEXT` protocol refers to a *cleartext* transmission, in that it deliberately avoids encryption. It is not an 'input to a cipher', as the term 'plaintext' might ordinarily imply.

Once the cleartext listener has been disabled, it is good practice to follow up with a corresponding firewall *deny* rule for all traffic inbound on port `9092`.

Broker-to-ZooKeeper encryption

While traffic between the Kafka brokers and the ZooKeeper ensemble can also be encrypted, the current version of Kafka (2.4.0 at the time of writing) does not support this feature natively. KIP-513[32] is targeting this for inclusion in release 2.5.0. To be clear, the limitation is specifically with the broker component of Kafka, not ZooKeeper. The latter supports mutual TLS.

[32] https://cwiki.apache.org/confluence/x/Cg6YBw

In the meantime, for the particularly security-minded deployments where encryption between brokers and the ensemble is a mandatory requirement, consider tunnelling the connection over a secure point-to-point link. This can be accomplished by positioning VPN terminators directly on the broker nodes, such that unencrypted traffic never leaves a broker. Alternatively, one can use a service mesh or a lightweight proxy capable of transparently initiating TLS connections, which are then natively terminated on ZooKeeper.

Encryption at rest

The encryption methods discussed earlier protect the confidentiality of data in transit. When data arrives at the broker, it will be persisted to its attached storage in indexed log segments — in cleartext.

Kafka does not have native facilities for enabling encrypted storage of record data. One has to resort to external options to accomplish this. The two popular approaches are full disk encryption and filesystem-level encryption. Both ensure that the disk is protected from unauthorized access when it is detached from the host.

While both forms of storage encryption provide a high level of protection against threats outside the host, the only way to protect the confidentiality of persisted data from embedded threats (rogue processes executing on the host) is to utilise end-to-end encryption. This involves encrypting outgoing records on the producer with either a shared-key or an asymmetric cipher — to be decrypted at the consumer end. This strategy ensures that neither the broker nor any intermediate conduits are aware of record contents. Kafka does not support end-to-end encryption natively. There is a provisional KIP-317[33] that discusses the prospect of adding this capability in the future. There are several open-source projects that implement this capability over the top of Kafka's serialization mechanism. One such example is the open-source *Kafka Encryption* project, hosted on GitHub at github.com/Quicksign/kafka-encryption[34].

When using end-to-end encryption, the information entropy of record batches approaches its theoretical maximum of unity. Therefore, it is best to disable compression, as the latter will only burn through CPU cycles without decreasing the payload size. (If anything, it will likely go up due to the overheads of compression.) For more information on compression, see Chapter 12: Batching and Compression.

End-to-end encryption, being focused solely on the record payload, does not eliminate the need for transport layer security. SSL (TLS) covers all aspects of the information exchange between clients and brokers, including metadata, record headers, offsets, group membership, and so on. SSL also protects the integrity of the data, guards against man-in-the-middle attacks, and uses X.509 certificates to provide assurance to the client party that the broker is authentic.

[33] https://cwiki.apache.org/confluence/x/AFIYBQ
[34] https://github.com/Quicksign/kafka-encryption

Authentication

Kafka supports several modes for attesting the identity of the connected clients. This section explores the authentication options at our disposal.

Mutual TLS

Mutual TLS (mTLS), also known as client-side X.509 authentication or two-way SSL/TLS, utilises the same principle of certificate signing used by conventional TLS, but in the opposite direction. In addition to the mandatory server-side authentication, the client presents a certificate that is verified by the broker. Each client will have a dedicated certificate, signed by a CA that is trusted by the broker. It may be the same CA used for the broker certificate signing, as is often the case.

To enable client authentication on the broker, one must set the `ssl.client.auth` property in `server.properties`. The permissible values are:

- `none`: Client authentication is disabled.
- `requested`: Client may optionally initiate SSL authentication, but this is not mandated by the broker. However, should the client present its certificate, it will be verified.
- `required`: The broker mandates the use of client-side SSL authentication. The client will not be allowed to connect without a valid certificate.

The `requested` option is a permissive 'halfway house' setting that enables the gradual migration to mTLS from unauthenticated TLS. To begin the transition, set all brokers to use `ssl.client.auth=requested`. This can be done statically — by editing each individual `server.properties` file and bouncing the brokers, or dynamically — via the admin API or the CLI tools. Once all brokers are running in permissive mode, begin the rollout of client updates. Once all clients have been verified to work stably with mTLS, upgrade the `ssl.client.auth` setting to `required`.

In order to use this authentication scheme, the client must be equipped with a dedicated private key, signed with a CA certificate that is trusted by the broker. The key and certificate should be installed in the `keystore.client.jks` file, alongside the existing `truststore.client.jks` file. The process below mimics the previous examples.

The following provides a worked example for enabling mutual TLS. Once again, the examples are going to use the `/tmp/kafka-ssl` directory. The `ca-key` and `ca-cert` files have been carried over from the previous examples.

Generate the private key

Generate a private key for the client application, using `localhost` as the value for the 'first and last name' attribute, leaving all others blank.

```
keytool -keystore client.keystore.jks -alias localhost \
    -validity 365 -genkey -keyalg RSA


Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  localhost
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]:
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]:
Is CN=localhost, OU=Unknown, O=Unknown, L=Unknown, □
    ST=Unknown, C=Unknown correct?
  [no]:  y

Generating 2,048 bit RSA key pair and self-signed
certificate (SHA256withRSA) with a validity of 365 days
  for: CN=localhost, OU=Unknown, O=Unknown, L=Unknown, □
      ST=Unknown, C=Unknown
```

This leaves a `client.keystore.jks` file in the current directory.

Sign the client certificate

```
keytool -keystore client.keystore.jks -alias localhost \
    -certreq -file client-cert-req
```

This produces the client certificate signing request file — `client-cert-req`.

Next, we will action the signing request with the existing CA:

```
openssl x509 -req -CA ca-cert -CAkey ca-key -in client-cert-req \
    -out client-cert-signed -days 365 -CAcreateserial
```

The result is the `client-cert-signed` file, ready to be imported into the client's keystore, along with the CA certificate.

Starting with the CA certificate:

```
keytool -keystore client.keystore.jks -alias CARoot \
    -import -file ca-cert


Enter keystore password:
Owner: CN=localhost
Issuer: CN=localhost
Serial number: 8f444e15ad8f7067
Valid from: Thu Jan 02 09:28:45 AEDT 2020 until: □
    Fri Jan 01 09:28:45 AEDT 2021
Certificate fingerprints:
   SHA1: 70:55:42:23:69:A1:EA:E8:13:49:41:CC:C3:CE:A3:□
      7B:CB:25:F8:08
   SHA256: 7E:CC:21:57:5B:8C:FB:90:D9:9E:2B:84:76:C4:□
      E1:83:D0:2D:B5:D1:17:3A:D2:D5:5A:4D:C5:CB:F3:9B:32:DD
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 1
Trust this certificate? [no]:  y
Certificate was added to keystore
```

Moving on to the signed certificate:

```
keytool -keystore client.keystore.jks -alias localhost \
    -import -file client-cert-signed


Enter keystore password:
Certificate reply was installed in keystore
```

Configure the broker to require authentication

Edit `server.properties`, adding the following line.

```
ssl.client.auth=required
```

Restart the server for the changes to take effect. To verify, you can run the producer client from the previous example. The connection should fail with the following error:

```
1:58:57/583   INFO  [kafka-producer-network-thread | producer-1]: □
    [Producer clientId=producer-1] Failed authentication with □
    localhost/127.0.0.1 (SSL handshake failed)
11:58:57/584   ERROR [kafka-producer-network-thread | producer-1]: □
    [Producer clientId=producer-1] Connection to node -1 □
    (localhost/127.0.0.1:9093) failed authentication due to: SSL □
    handshake failed
```

That is to be expected; our SSL-enabled client has yet to be configured for client-side authentication.

Deploy the private key and signed certificate to the client

```
cp /tmp/kafka-ssl/client.keystore.jks \
    ~/code/effectivekafka
```

Configure the client to provide authentication

Enabling mutual authentication on the client requires the addition of several properties — namely, the SSL keystore configuration. In the snippet below, we have provided the `ssl.keystore.location`, `ssl.keystore.password` and `ssl.key.password` — being the mirror image of the server-side configuration.

```
final var config = new HashMap<String, Object>();
config.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
           "localhost:9093");
config.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG,
           "SSL");
config.put(SslConfigs.SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG,
           "https");
config.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,
           "client.truststore.jks");
config.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG,
           "secret");
config.put(SslConfigs.SSL_KEYSTORE_LOCATION_CONFIG,
           "client.keystore.jks");
config.put(SslConfigs.SSL_KEYSTORE_PASSWORD_CONFIG,
           "secret");
config.put(SslConfigs.SSL_KEY_PASSWORD_CONFIG,
           "secret");
config.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
           StringSerializer.class.getName());
config.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
           StringSerializer.class.getName());
```

```
config.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,
           true);

// ... the existing producer code
```

Identifying the user principal

When authenticating a client to the broker, hostname verification is not performed. Instead, the user principal is taken verbatim as the value of the CN attribute of the certificate. One can change how a certificate's attributes translate to a username, by overriding the `ssl.principal.mapping.rules` property to provide an alternate mapping. The format of this setting is a list, where each rule starts with the prefix `RULE:`, and specifies a regular expression to match the desired attributes, followed by a `/` (slash) character, then by a replacement, another slash, and finally, by an optional `U` or `L` character indicating whether the replacement should be capitalised or forced to lower case, respectively. The format is depicted below.

```
RULE:pattern/replacement/
RULE:pattern/replacement/[LU]
```

There are several examples listed on the official Kafka documentation page at kafka.apache.org/documentation[35].

Mutual TLS is frequently used in machine-to-machine authentication — ideal for the typical Kafka use case, where the client party is an application that operates independently of its end-users. There may be complications, however. The degree of authentication assurance in mTLS is limited by the level of rigour present in the certificate signing process. Specifically, the broker can determine with a high level of assurance that the client is a *trusted party*. But which trusted party? That is the more challenging question.

Recall, the client is trusted because it presents a valid certificate signed by a trusted issuer. The username implied by the certificate (represented by the CN by default) is assumed to be strongly associated with the party that presented the certificate, because this relationship was presumably independently verified by the issuer — a CA that we trust *a priori*.

Consider, for example, two trusted clients: *C0* and *C1*. Both have a dedicated private key, signed by a trusted CA for use with a specific CN. However, if the signing process is naively implemented, there is little stopping *C0* from issuing another signing request to the CA, using the username of *C1* in the CN field. If this were to happen, *C0* could connect to the broker with an alternate certificate — impersonating *C1*. In order to prevent this from occurring, the process of signing certificates cannot be left entirely in the clients' hands — irrespective of whether the clients operate within the organisation's perimeter. The relationship between a client and the CA must be individually authenticated, so that a client can only issue a signing request for a CN that has been authorized for that client.

[35] https://kafka.apache.org/documentation/#security_authz_ssl

> A layered approach to information security — the use of principles such as *Defence in Depth* — imply that an organisation should trust internal entities little more than it should trust external ones. Recent studies have indicated that as much as 60% of attacks originate from within the perimeter. And while PKI and public-key cryptography allow us to reason concretely as to the authenticity or the integrity of data, they are predicated on assumptions that are often taken for granted.

Another challenge of using mTLS is that the identity of clients is not known in advance, as there is no requirement to explicitly enrol each user principal into Kafka. On one hand, this clearly simplifies identity generation by decoupling it from the authentication mechanism. On the other hand, the management of identities is significantly more complicated than username/password-based methods, as Kafka does not support certificate revocation. In other words, if the private key of one client (or a small group of clients) becomes compromised, there is no way to instruct the brokers to blacklist the offending certificates. At best, one would have to deploy a new CA certificate in place of the compromised one and sign all remaining legitimate client certificates with the novated CA.

Technically, although certificates cannot be revoked, one can remove all privileges from the affected user principals, and require the rotation of the affected usernames. Frankly, this cannot be called a definitive solution — more of a workaround. Although username rotation fixes the immediate threat, it does so at the authorization layer, which is distinct from authentication. There is still a threat that the original user's permissions might be accidentally restored later, at which point the compromised key will become a threat again. Without regenerating the CA certificate, there is no way to *permanently* disable a user.

Certificate revocation has been an open issue in Kafka since May 2016. As such, avoid the use of mutual TLS in environments where client deployments are not secure, and where there is a likelihood of a key compromise — even if the client acts as a low-privilege user on a test system. While the implication a single compromise might not be catastrophic, the liquidation of the consequences will be long and laborious. To that point, when using mTLS with multiple Kafka clusters (e.g. multiple environments or geographical sites), it is best to segregate trust chains, so that each cluster trusts its own intermediate CA certificate but not the others', and clients are selectively issued with certificates that are individually signed for the specific clusters that they are legitimately expected to access.

Mutual TLS is used frequently for authenticating clients to managed Kafka providers. The service consumer will typically provide an intermediate CA certificate to the service provider, which then allows the service consumer to deploy any number of Kafka clients with dedicated certificates.

Despite operating at different layers of the OSI model and being independent in theory, two-way SSL cannot be used in conjunction with application-level authentication schemes. In Kafka, it is either one or the other.

SASL

Kafka supports *Simple Authentication and Security Layer* (SASL) — an extensible framework for embedding authentication in application protocols. The strength of SASL is that it decouples authentication concerns from application protocols, in theory allowing any authentication mechanism supported by SASL to be embedded in any application that uses SASL. SASL is not responsible for the confidentiality of the message exchange, and is therefore rarely used on its own. The most common deployment model of SASL involves TLS.

GSSAPI

Kafka supports the Generic Security Service API (GSSAPI), which enables compliant security-service vendors to exchange opaque tokens with participating applications. These tokens are internally tamper-proof, and are used to establish a security context among collaborating parties. Once the context is established, the parties can communicate securely and access resources.

GSSAPI is commonly associated with Kerberos, being its dominant implementation. The primary distinction is that the Kerberos API has not been standardised among vendors, whereas GSSAPI has been vendor-neutral from the outset. In a manner of speaking, GSSAPI standardises Kerberos.

The official Kafka documentation treats GSSAPI synonymously with Kerberos. The professed relationship between GSSAPI and Kerberos typically extends to centralised directory services — Active Directory (predominantly due to its native Kerberos version 5 support) and similarly positioned corporate single sign-on (SSO) services.

Kerberos and Active Directory work best for interactive users in a corporate setting; however, Kafka users are rarely individuals, but applications. Even when a Kafka client acts on behalf of an end-user, it will ordinarily authenticate itself to the broker via an end-user-agnostic mechanism — typically a *service account*. The authenticated session outlives any single user interaction; due to their time and resource costs, it is generally infeasible to spawn a new set of connections for every user interaction, only to tear them down shortly afterwards. In the overwhelming majority of cases, service accounts are the only practical way to manage Kafka clients.

In theory, a service account could be provisioned in a centralised corporate authentication server such as Active Directory. On the surface, it even sounds beneficial. By centralising service accounts, it should make their governance a more straightforward and transparent affair.

While the theoretical advantages are not disputed, the main challenge with the centralised model is that it only works if the principal in question can be modelled as a resource in a directory and all the resources it consumes are represented in some sort of a permission model that can be associated with the principal. In other words, an application can access all of its resources using a single set of credentials. (Or these credentials are somehow related, and their relationship can be modelled accordingly.) In reality, backend application components may consume numerous disparate resources,

not all of which may be enrolled in the directory. For example, a single application may connect to Kafka, a Postgres database, a Redis cluster, and may consume third-party APIs that are external to the organisation. While Kafka and Postgres support Kerberos, Redis does not. And a third-party API will almost certainly not support Kerberos. (Some service providers may indirectly support Kerberos, typically via a federation protocol.)

When the time comes to control the permissions of a service account, the administrator might manipulate the representation of the service account in the directory. Perhaps the service account should be disabled altogether, which is a trivial 'one-click' action in a directory service such as Active Directory. This creates a false sense of security, as the permissions for the service account do not cover all resources; while the service account may appear to be disabled in the centralised directory, the underlying principal is still able to operate on a subset of its resources.

> It is a common misunderstanding that centralised authentication systems exist to simplify the on-boarding process. In practice, the benefits of such systems are predominantly concentrated in the off-boarding scenario. It is convenient to quickly set up a profile for a new employee with one click of a button, granting them access to all necessary systems. Convenient — but not essential. Without a centralised authentication system, the on-boarding process may take days or weeks to complete. And while this is not ideal, it is mostly the productivity loss that affects the organisation — a cost that can be quantified and managed accordingly. Conversely, termination of employment requires immediate action from a security standpoint — revoking the employee's access simultaneously from all resources previously available to them, preferably before the person has left the premises. Failure to act in a timely manner, or forgetting to apply the change to some resources, may pose an immediate and persistent threat to the organisation, particularly if the employee was not completely content with the circumstances of their dismissal.
>
> While the benefits of centralised authentication are clear for end-users and administrators, particularly in a corporate setting, the equivalent benefits cannot be easily reproduced for service accounts — not unless the organisation mandates that all technical infrastructure supports Kerberos and all interactions between application components and their dependencies are authenticated accordingly. Stated plainly, this is a pipe dream.
>
> For service accounts and similar integration use cases, the prevalent industry trend is in the migration away from centralised directory-based authentication systems towards centralised secrets management systems. In other words, rather than managing principals, which we established is futile, we manage their credentials and any other secret material they need to function. Provided the principal receives the entirety of its credentials from the secrets management system and does not cache them locally, then disabling the principal will *eventually* affect its ability to consume downstream resources, effectively isolating it.
>
> One notable drawback of this scheme is that the effect of disabling a principal may not be timely. While it makes future retrieval of secret material impossible, it does not necessarily render the existing material invalid or obsolete. On top of this, the client may have cached the secret material or the latter may have become compromised by other means. This is why security best-practices

> suggest *frequent rotation of secret material*, down to the order of minutes — the time-exposure of a sensitive artifact is thereby minimised.

This chapter does not delve further into the authentication cases for GSSAPI, not solely because of the statements above. Rather, the benefit of demonstrating Kerberos is not worth the complexity of setting it up. It is also adequately documented at kafka.apache.org/documentation[36]. Instead, the focus will shift towards other SASL authentication methods that are less draconian, more straightforward to administer, and are likely to remain relevant to the reader for many years to come.

PLAIN and SCRAM

The complete source code listings for the SASL producer and consumer client examples are available at github.com/ekoutanov/effectivekafka[37] in the `src/main/java/effectivekafka/sasl` directory.

SASL offers two username/password-based authentication modes: PLAIN and SCRAM. (There are more in existence, but Kafka just supports these two for authenticating clients.) The former refers to cleartext authentication where credentials are presented verbatim. The latter is an acronym for *Salted Challenge Response Authentication Mechanism* — a protocol designed to fulfil authentication without the explicit transfer of credentials.

To the user, there is little discernible difference between the two options. In both cases, the application is configured with a username and password pair, with the only difference being the fully-qualified class name of the JAAS (Java Authentication and Authorization Service) module. Under the hood, the two are very different. PLAIN provides no confidentiality of its own, requiring the use of an encrypted channel. SCRAM is secure in its own right, providing confidentiality in the absence of any transport layer encryption.

> In case the question might arise, PLAIN is not an acronym. The exact origin of the name is unspecified, but it is likely a candid capitalisation of the adjective 'plain' — being "simple or basic in character", as opposed to an abbreviation of 'plaintext' — the cryptographic term meaning "input to a cipher". Indeed, RFC 4616[a] specifies PLAIN as a *simple mechanism for the exchange of passwords*. The exchange always occurs in cleartext — there is no intention of encryption at the application level. As such, PLAIN is intended to be used in concert with lower-level mechanisms that guarantee the confidentiality of data. Quoting from the RFC:
>
> > As the PLAIN mechanism itself provides no integrity or confidentiality protections, it

[36] https://kafka.apache.org/documentation/#security_sasl_kerberos
[37] https://github.com/ekoutanov/effectivekafka/tree/master/src/main/java/effectivekafka/sasl

> should not be used without adequate external data security protection, such as TLS services provided by many application-layer protocols. By default, implementations **should not** advertise and **should not** make use of the PLAIN mechanism unless adequate data security services are in place.

[a] https://tools.ietf.org/html/rfc4616

Guided by the *Defence in Depth* principle, the SCRAM option should be preferred over PLAIN. This ensures that no control will singlehandedly impact the integrity of the system if compromised. SCRAM acts in both directions: not only must the client prove to the broker that it has the password, but the broker is required to do the same. (Caveat below.) In other words, SCRAM protects the client from connecting to a rogue broker, which acts in addition to the certificate-based attestation mechanism used natively in TLS. The only practical drawback of SCRAM over PLAIN is the added network round trip, which is necessary for both parties to identify one another. (SCRAM requires a total of two round-trips.) However, this penalty is paid once, during connection establishment — it has no subsequent bearing on throughput or latency.

Although SCRAM authentication is bidirectional, it is not completely symmetric — there is a subtle difference between the assurance each party provides to the other. Specifically, the client must prove to the broker that it has *present* knowledge of the password — at the time of connection establishment; whereas the broker only needs to prove that it knew the password at some point in time. This is a desirable property — it relieves the broker from having to persist the password verbatim, and therefore risking exposure; instead, the broker persists irreversible derivations of the password that are subsequently used for mutual authentication. In theory, the client can discard the password and store the hashed version, provided it knows the salt used on the server (which is disclosed as part of the authentication flow).

Because of the similarity in how the two authentication methods appear to the user, the worked examples will focus on the SCRAM case.

Configure the broker to use SASL/SCRAM

We need to create a new listener that supports the use of SASL authentication. Edit `server.properties`, ensuring that it is in line with the following:

```
listeners=PLAINTEXT://:9092,SSL://:9093,SASL_SSL://:9094
advertised.listeners=PLAINTEXT://localhost:9092,\
    SSL://localhost:9093,SASL_SSL://localhost:9094
listener.security.protocol.map=PLAINTEXT:PLAINTEXT,\
    SSL:SSL,SASL_PLAINTEXT:SASL_PLAINTEXT,SASL_SSL:SASL_SSL
sasl.enabled.mechanisms=SCRAM-SHA-512
listener.name.sasl_ssl.scram-sha-512.sasl.jaas.config= \
    org.apache.kafka.common.security.scram.ScramLoginModule \
    required;
```

Having saved the file, restart the broker for the change to take effect.

Working through the above file, line-by-line:

- We added another listener named `SASL_SSL`, bound to port `9094`.
- The listener has been advertised as `SASL_SSL://localhost:9094`.
- The security protocol map has been left as is, given it already contains an `SASL_SSL:SASL_SSL` mapping out of the box.
- The enabled SASL mechanism is `SCRAM-SHA-512`.
- The `listener.name.sasl_ssl.scram-sha-512.sasl.jaas.config` property provides a minimal JAAS document that mandates the use of the `ScramLoginModule`.

SASL authentication is incompatible with SSL client authentication. When the `ssl.client.auth` property is set to `requested` or `required`, the SSL authentication settings will only be applied to the `SSL` listener (or any listener mapped to the `SSL` security protocol). When connecting over the `SASL_SSL` listener, the SSL client authentication settings will be ignored. It would have been advantageous to run SASL on top of SSL client authentication for added security; and while this is possible in theory, Kafka does not support the conjunction of the two. The reason is that Kafka extracts the user principal from the attributes of the client certificate, which may conflict with the username provided over SASL.

Kafka supports `SCRAM-SHA-256` and `SCRAM-SHA-512`. Both can be enabled if need be, by setting the `sasl.enabled.mechanisms` broker property to `SCRAM-SHA-256,SCRAM-SHA-512`. Security-wise, both SHA-256 and SHA-512 are generally considered to be very strong, with the latter having better collision resistance. Different hardware favours different functions, with SHA-512 optimised for use on 64-bit processors; SHA-256 being more performant on 32-bit processors. SHA-256 remains the more common choice for the time being.

Provision the user

Kafka's SCRAM implementation uses ZooKeeper as a credential store — for persisting usernames and hashed passwords (which are also salted). Cleartext passwords are never persisted, but the

hashes are world-readable. This implies that ZooKeeper should be deployed on a highly trusted network that is protected from access by unauthorised parties. Configuration is administered directly against ZooKeeper, using the `kafka-configs.sh` CLI. In the following example, we will provision a new user with the username `alice`, the password being `alice-secret`.

```
$KAFKA_HOME/bin/kafka-configs.sh --zookeeper localhost:2181 \
    --alter \
    --add-config 'SCRAM-SHA-512=[password=alice-secret]' \
    --entity-type users --entity-name alice


Completed Updating config for entity: user-principal 'alice'.
```

When enrolling credentials into ZooKeeper, they need to be specified separately for each variation of the SCRAM algorithm. In other words, `SCRAM-SHA-256` credentials may differ from `SCRAM-SHA-512`, and can be administered independently. The `kafka-configs.sh` CLI conveniently supports specifying lists of key-value pairs in the `--add-config` flag.

To list the current configuration for a user, run the following (replacing the username as appropriate):

```
$KAFKA_HOME/bin/kafka-configs.sh --zookeeper localhost:2181 \
    --describe --entity-type users --entity-name alice
```

This will output the current configuration:

```
Configs for user-principal 'alice' are SCRAM-SHA-512=salt= □
    ZWhuNHk2b2ZxaHRxcHQxamlhdDYzbzg=,stored_key=hpRpfIoJZc □
    Orhy9/6Fh80VsBhNCKKpMTZtWKkCs08H8us15pphr5upfTGjvGYYON □
    EeDfw002WUqfT6g+TRJqA==,server_key=y8yhtDe20lH2hK9a6tk □
    eidFTL2A3E4KV+Pd6U5QfXc5ndj0nzzfl5N1xX0m5W4EVccDphXIDG □
    T5wtuEAAnmv5g==,iterations=4096
```

Note that the output of `--describe` is different to what was fed as the input to `--add-config`. This is because SCRAM persists derived values of the password. These values can be used to cryptographically verify the mutual knowledge of the password during an authentication session, but the original password cannot be recovered using this method.

Credentials may be deleted using the `--delete-config` flag:

```
$KAFKA_HOME/bin/kafka-configs.sh --zookeeper localhost:2181 \
    --alter \
    --delete-config 'SCRAM-SHA-512'\
    --entity-type users --entity-name alice
```

Configure the client

Configuring a client to use SASL requires just a few additions to the configuration map. The example below is a fully-functional producer that uses a combination of SSL and SASL.

```
import static java.lang.System.*;

import java.util.*;

import org.apache.kafka.clients.*;
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.config.*;
import org.apache.kafka.common.security.scram.*;
import org.apache.kafka.common.serialization.*;

public final class SaslSslProducerSample {
  public static void main(String[] args)
      throws InterruptedException {
    final var topic = "getting-started";

    final var loginModuleClass = ScramLoginModule.class.getName();
    final var saslJaasConfig = loginModuleClass
        + " required\n"
        + "username=\"alice\"\n"
        + "password=\"alice-secret\";";

    final Map<String, Object> config = Map
        .of(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "localhost:9094",
            CommonClientConfigs.SECURITY_PROTOCOL_CONFIG,
            "SASL_SSL",
            SslConfigs.SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG,
            "https",
            SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,
            "client.truststore.jks",
            SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG,
            "secret",
            SaslConfigs.SASL_MECHANISM,
```

```
            "SCRAM-SHA-512",
            SaslConfigs.SASL_JAAS_CONFIG,
            saslJaasConfig,
            ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName(),
            ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName(),
            ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,
            true);

    try (var producer = new KafkaProducer<String, String>(config)) {
      while (true) {
        final var key = "myKey";
        final var value = new Date().toString();
        out.format("Publishing record with value %s%n",
                   value);

        final Callback callback = (metadata, exception) -> {
          out.format("Published with metadata: %s, error: %s%n",
                     metadata, exception);
        };

        // publish the record, handling the metadata in the callback
        producer.send(new ProducerRecord<>(topic, key, value),
                      callback);

        // wait a second before publishing another
        Thread.sleep(1000);
      }
    }
  }
}
```

The differences from the pure-SSL example are in the following:

- Connection to a different port. In this case, we are connecting to port `9094`.
- Use of `SASL_SSL` for the `security.protocol` setting.
- Addition of the `sasl.mechanism` property, set to `SCRAM-SHA-512`.
- Specifying the JAAS configuration via the `sasl.jaas.config` property.

The JAAS configuration for example above is:

```
org.apache.kafka.common.security.scram.ScramLoginModule required
    username="alice"
    password="alice-secret";
```

Interbroker authentication

Although we changed client authentication to `SASL_SSL`, the interbroker listener remained `SSL` from the first example. We can upgrade interbroker communications to use authentication in addition to encryption; however, this requires us to configure a set of credentials specifically for broker use.

Start by creating a set of admin credentials (replace `admin` and `admin-secret` as appropriate):

```
$KAFKA_HOME/bin/kafka-configs.sh --zookeeper localhost:2181 \
    --alter \
    --add-config 'SCRAM-SHA-512=[password=admin-secret]' \
    --entity-type users --entity-name admin
```

Then edit `server.properties` to line up with the following. (Only the changed lines are shown.)

```
inter.broker.listener.name=SASL_SSL
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-512
listener.name.sasl_ssl.scram-sha-512.sasl.jaas.config= \
    org.apache.kafka.common.security.scram.ScramLoginModule \
        required \
        username="admin" \
        password="admin-secret";
```

This is where we need to be careful. Having specified credentials in `server.properties`, we *should* harden the file permissions to prevent the file from being read by anyone other than the currently logged in user. If this user is different from the user that will run Kafka, then the ownership and permissions should be adjusted accordingly.

```
chmod 600 $KAFKA_HOME/config/server.properties
```

Having saved the file and adjusted its permissions, restart the broker for the change to take effect.

Summary of the changes:

- The interbroker listener was changed to `SASL_SSL`.
- The SASL mechanism for interbroker use was assigned via the `sasl.mechanism.inter.broker.protocol` property.
- The JAAS configuration defined in `listener.name.sasl_ssl.scram-sha-512.sasl.jaas.config` was expanded to include a username and password pair.

Having switched the interbroker protocol, run `netstat` to verify that interbroker connections are directed at port `9094`:

```
netstat -an | egrep "9092|9093|9094"
```

```
tcp4    0   0  127.0.0.1.9094    127.0.0.1.56410   ESTABLISHED
tcp4    0   0  127.0.0.1.56410   127.0.0.1.9094    ESTABLISHED
tcp46   0   0  *.9094            *.*               LISTEN
tcp46   0   0  *.9093            *.*               LISTEN
tcp46   0   0  *.9092            *.*               LISTEN
```

Supplying an external JAAS configuration

In previous SASL examples, we supplied the JAAS configuration in-line, via a configuration property. And while this approach is convenient, it may not always be appropriate — depending on whether the application supports this method of configuration. Where in-line configuration is not supported, JAAS can be configured using an external file.

To alter the previous example to use an external JAAS configuration file, edit `server.properties` and comment out (or remove) the `listener.name.sasl_ssl.scram-sha-512.sasl.jaas.config` property. Save the file.

If you attempt to start the broker in this state, it will bail out with the following error:

```
[2020-01-02 19:22:25,890] ERROR [KafkaServer id=0] Fatal error ⮒
    during KafkaServer startup. Prepare to shutdown ⮒
    (kafka.server.KafkaServer) ⮒
    java.lang.IllegalArgumentException: Could not find a ⮒
    'KafkaServer' or 'sasl_ssl.KafkaServer' entry in the ⮒
    JAAS configuration. System property ⮒
    'java.security.auth.login.config' is not set
```

To fix this error, create a `kafka_server_jaas.conf` file in `$KAFKA_HOME/config`, with the following contents:

```
sasl_ssl.KafkaServer {
    org.apache.kafka.common.security.scram.ScramLoginModule
        required
        username="admin"
        password="admin-secret";
};
```

The JAAS configuration contains sensitive information. Ensure it is not readable by other users by running the following:

```
chmod 600 $KAFKA_HOME/config/kafka_server_jaas.conf
```

When starting the Kafka broker, it needs to be told where to find the JAAS configuration. This can be accomplished by setting the `java.security.auth.login.config` system property, which is passed via the `KAFKA_OPTS` environment variable, as shown below.

```
JAAS_CONFIG=$KAFKA_HOME/config/kafka_server_jaas.conf \
KAFKA_OPTS=-Djava.security.auth.login.config=$JAAS_CONFIG \
$KAFKA_HOME/bin/kafka-server-start.sh \
    $KAFKA_HOME/config/server.properties
```

Switching to SASL/PLAIN

It was previously stated that SASL/PLAIN and SASL/SCRAM are not materially differentiated from a user's perspective. To change clients over to SASL/PLAIN, set the `sasl.mechanism` client property to `PLAIN` and replace the fully-qualified class name of the JAAS module from `org.apache.kafka.common.security.sc` to `org.apache.kafka.common.security.plain.PlainLoginModule`.

On the broker, the differences are slightly more perceptible. PLAIN mode does not use ZooKeeper to store credentials. Instead, credentials are defined directly in the JAAS configuration.

To enable SASL/PLAIN, edit `server.properties`, changing the following lines:

```
sasl.enabled.mechanisms=PLAIN
sasl.mechanism.inter.broker.protocol=PLAIN
listener.name.sasl_ssl.plain.sasl.jaas.config= \
    org.apache.kafka.common.security.plain.PlainLoginModule
        required \
        username="admin" \
        password="admin-secret" \
        user_admin="admin-secret" \
        user_alice="alice-secret";
```

Also, remove the `listener.name.sasl_ssl.scram-sha-512.sasl.jaas.config` property from the SASL/SCRAM example. (Unless you need to maintain SCRAM alongside PLAIN.)

On the client-side, change the value of the `sasl.mechanism` property to `PLAIN`.

OAuth bearer

The `OAUTHBEARER` SASL mechanism enables the use of OAuth 2.0 Access Tokens to authenticate user principals. *The primary motivation for OAuth support is not for security*, but for ease of integration and testing, allowing applications to impersonate users by way of an *Unsecured JWS* token in non-production environments. In other words, an application can pose as an arbitrary user by issuing an unsigned JWT (JSON Web Token), where the header does not specify an algorithm:

```
{
  "alg": "none"
}
```

The JWS Signature used for unsecured tokens is an empty octet string, which base-64 encodes simply to an empty string.

Unsecured tokens function out-of-the-box with minimal configuration and with no OAuth 2.0 infrastructure required. The principal's username is taken directly from the `sub` (subject) claim in the JWT.

In its default guise, the `OAUTHBEARER` mechanism cannot be used securely in production environments, as it cannot verify user claims. In order to productionise this algorithm, one must implement a pair of callback handlers. The callbacks are necessary to allow the client to generate and sign access tokens, and for the broker to validate them against a trusted certificate.

On the client, implement the `org.apache.kafka.common.security.auth.AuthenticateCallbackHandler` interface to retrieve a token. The implementation must be able to handle an instance of an `OAuthBearerTokenCallbac` feeding it a well-formed `OAuthBearerToken`. The implementation class is specified via the `sasl.login.callback.hand` client property.

On the broker, implement the `org.apache.kafka.common.security.auth.AuthenticateCallbackHandler` interface capable of handling an `OAuthBearerValidatorCallback`. Its role is to validate the attributes of the token and feed a validated token to the callback. The implementation class is configured via the `listener.name.sasl_ssl.oauthbearer.sasl.server.callback.handler.class` property. The class definition, along with all dependencies must be on the classpath of the broker's JVM instance. This can be accomplished by packaging the callback handler and its dependencies into a Jar file, and placing the latter into `$KAFKA_HOME/libs`.

Rather than implementing OAuth handling from scratch, you can use the open-source *Kafka OAuth* project, located at github.com/jairsjunior/kafka-oauth[38].

A secondary objective of supporting OAuth bearer tokens was the internal testing of SASL. Kafka's SASL implementation can now be considered mature, but nonetheless, the `OAUTHBEARER` can still be useful for testing with different users.

Delegation tokens

Concluding the discussion on authentication methods, we have *delegation tokens*. Delegation tokens are used as a lightweight authentication mechanism that is complementary to SASL. It was introduced in KIP-48[39] as part of release 1.1.0. The initial motivation for delegation tokens was to simplify

[38] https://github.com/jairsjunior/kafka-oauth
[39] https://cwiki.apache.org/confluence/x/tfmnAw

the logistics of key distribution for Kerberos clients — specifically, the need to deploy TGTs or keytab files to each client. Delegation tokens also reduce the overhead of the authentication process — rather than engaging the KDC to get a ticket and periodically renewing the TGT, the authentication protocol is reduced to verifying the delegation token, which is persisted in ZooKeeper. This approach is also more secure, reducing the blast radius of compromised credentials — if a delegation token is covertly obtained, the attacker is limited to the permissions attached to the delegation token. By comparison, the compromise of a Kerberos TGT or keytab carries more dire consequences.

The most common use case for delegation tokens is event stream processing where a swarm of worker nodes is orchestrated by a centralised coordinator. Worker nodes tend to be ephemeral. Rather than provisioning access independently for each worker node, the coordinator creates a time-bounded delegation token, then spins up a worker node, handing it the newly-generated token. Under this scheme, only the coordinator node requires access to long-lived credentials (a TGT or keytab for Kerberos, or a username/password pair for SCRAM or PLAIN); the highly sensitive material never leaves the coordinator.

Delegation tokens are issued for a finite lifespan — an upper bound on the maximum age of a token. Furthermore, a token has an expiration time, which sets a soft limit on the time that the token may be used to authenticate a client. The expiry time may be extended by renewing the token, subject to the upper bound enforced by its lifespan. An expired token is eventually purged from ZooKeeper using a background housekeeping thread. Because purging happens asynchronously, it is possible for a token to be used for some time after its official expiry.

Enable delegation tokens on the broker

To enable delegation tokens, edit `server.properties`, adding the following lines:

```
delegation.token.master.key=secret-master-key
delegation.token.expiry.time.ms=3600000
delegation.token.max.lifetime.ms=7200000
```

The `delegation.token.master.key` property specifies a key for signing tokens that must be shared by all brokers in a cluster. This property is required — delegation tokens are disabled without it. Replace the string `secret-master-key` as appropriate.

The `delegation.token.expiry.time.ms` specifies the token expiry time in milliseconds. The default value is `86400000` (24 hours).

The `delegation.token.max.lifetime.ms` specifies the hard upper bound on the lifespan of the token in milliseconds. When issuing a token, the user can specify any value for the `--max-life-time-period` that is below this setting. The default value of `delegation.token.max.lifetime.ms` is `604800000` (7 days).

Restart the broker for the changes to take effect.

Creating delegation tokens

To create a delegation token, use the `kafka-delegation-tokens.sh` CLI, as shown in the example below.

```
$KAFKA_HOME/bin/kafka-delegation-tokens.sh \
    --bootstrap-server localhost:9094 \
    --command-config $KAFKA_HOME/config/client.properties \
    --create --max-life-time-period -1 \
    --renewer-principal User:admin
```

The owner of the token is the principal specified in the supplied properties file. In our example, `client.properties` has been configured with the credentials of the `admin` user. This is not ideal in production, as it gives the token bearer significantly more privileges than it likely requires.

Running the above command produces the following output:

```
Calling create token operation with renewers : □
    [User:admin] , max-life-time-period :-1
Created delegation token with tokenId : □
    doFHaIYjQwWhyeBZrBW54w

TOKENID
doFHaIYjQwWhyeBZrBW54w
 □
HMAC
AFtqnd/cV/fFCawRYhPIHqe9sLZGegYscYu1o8BwSDn11 □
    rIqjKWRsKyLS9+CJ5Jor4RsAckTcS0IAWvdAlPqjQ==
 □
OWNER                RENEWERS
User:admin           [User:admin]
 □
ISSUEDATE            EXPIRYDATE           MAXDATE
2020-01-02T11:43     2020-01-01T12:43     2020-01-03T11:43
```

The `--max-life-time-period` flag specified the maximum lifespan of the token in milliseconds. If set to `-1`, the maximum admissible value specified by the `delegation.token.max.lifetime.ms` broker property is assumed.

The `--renewer-principal` flag specifies the user who is allowed to renew the token. This user may be different from the owner. When renewing the token, the user must present their credentials in addition to the token's HMAC value.

To list delegation tokens, invoke `kafka-delegation-tokens.sh` with the `--describe` switch:

```
$KAFKA_HOME/bin/kafka-delegation-tokens.sh \
    --bootstrap-server localhost:9094 \
    --command-config $KAFKA_HOME/config/client.properties \
    --describe
```

Kafka presently does not allow one user to create a delegation token on behalf of another (where the owner principal is different to the maker of the token). KIP-373[40] is slated to address this shortfall.

Configuring clients

Delegation tokens are passed in similarly to credentials, using the SASL/SCRAM method. The `username` attribute is set to the token ID, while the `password` attribute is assigned the `HMAC` value from the table above. In addition to these attributes, a third attribute `tokenauth` must be present, set to `true`. This attribute distinguishes token authentication from conventional username/password authentication.

```
final var saslJaasConfig = loginModuleClass
    + " required\n"
    + "username=\"doFHaIYjQwWhyeBZrBW54w\"\n"
    + "password=\"AFtqnd/cV/fFCawR...PqjQ==\"\n"
    + "tokenauth=\"true\";";
```

Renewing tokens

The renewal of a token can only be attempted by the user listed in `--renewer-principal`, and must happen before the token expires. The `kafka-delegation-tokens.sh` command requires both the user's credentials and the token being renewed — specifically, the `HMAC` value:

```
$KAFKA_HOME/bin/kafka-delegation-tokens.sh \
    --bootstrap-server localhost:9094 \
    --command-config $KAFKA_HOME/config/client.properties \
    --renew --renew-time-period -1 \
    --hmac AFtqnd/cV/fFCawR...PqjQ==
```

The `--renew-time-period` flag specifies the duration of time (in milliseconds) that the token should be extended by. If set to `-1`, the token will be renewed for the duration specified by the `delegation.token.expiry.tim` broker property.

Expiring tokens

The expiration time of a token can be adjusted following its creation. The most common case is to explicitly invalidate a token, accomplished using the command below.

[40] https://cwiki.apache.org/confluence/x/cwOQBQ

```
$KAFKA_HOME/bin/kafka-delegation-tokens.sh \
    --bootstrap-server localhost:9094 \
    --command-config $KAFKA_HOME/config/client.properties \
    --expire --expiry-time-period -1 \
    --hmac AFtqnd/cV/fFCawR...PqjQ==
```

The `--expiry-time-period` switch specifies an extension on the expiration time in milliseconds. If set to `-1`, the token will be expired immediately.

Rotating secrets

Kafka does not yet have an elegant mechanism for rotating the shared secret. This is currently a three-step process:

1. Expire all existing tokens.
2. Perform a rolling bounce of the cluster with the new secret.
3. Generate and distribute new tokens.

As tokens are verified only during connection establishment, any clients that are already connected will continue to function normally. New connections using old tokens will be rejected, as will any attempt to renew or expire tokens.

ZooKeeper authentication

The final stretch in our authentication journey is hardening the link between the brokers and the ZooKeeper ensemble. ZooKeeper supports SASL client authentication using the DIGEST-MD5 method. (In our context, the client is the Kafka broker or a CLI tool.)

> ZooKeeper authentication powers its authorization model, which is implemented by way of an Access Control List (ACL). An ACL applies individually to each znode, in a manner that is similar to UNIX file and directory permissions.
>
> ZooKeeper supports the following permission types on znodes:
>
> - `CREATE`: create a child node.
> - `READ`: get data from a node and list its children.
> - `WRITE`: set data for a node.
> - `DELETE`: delete a child node.
> - `ADMIN`: set permissions.
>
> Permissions are recorded against a znode in a list. Each element is a triplet, comprising the authentication scheme, the allowed principal, and the set of permissions. There are several built-in schemes. The two notable ones that apply in our context are `world` — meaning anyone (including unauthenticated users), and `sasl` — referring to a user that has been authenticated via SASL.

> The version of ZooKeeper (3.5.6) bundled with the latest Kafka version (2.4.0 at the time of writing) does not enforce authentication — only authorization. A client can connect without presenting any credentials — its session will be associated with the `world` scheme. Although we can't block unauthenticated connections, we can limit their level of access using znode ACLs. In effect, this is how ZooKeeper restricts unauthenticated connections — it accepts the connection but restricts their further actions.

Enabling authentication on a ZooKeeper ensemble that has already been confined to a segregated network seems excessive. And in some ways, it is. The security of information assets can be compounded almost indefinitely until the usability of the system is crippled for legitimate users. At which point one asks: *Have we just made the system less secure by blurring the line between legitimate and illegitimate users?*

The answer to this is subjective, and it really depends on several factors, such as existing security controls, the nature of the industry and level of regulation, existing organisational security policies and guidelines, the likelihood of a breach and the cost of mitigation. The following subsections include worked examples for enabling ZooKeeper authentication for new and existing clusters; however, the rest of the chapter and the book will assume that ZooKeeper authentication is disabled.

Enable authentication on ZooKeeper

ZooKeeper supports a different set of SASL methods compared to what Kafka offers its clients; nonetheless, the basic principles are the same. Like Kafka, ZooKeeper uses JAAS for configuration.

Create a new file named `zookeeper_jaas.conf` in the `$KAFKA_HOME/config` directory, with the contents below. The admin username will be `zkadmin`. Replace the value `zkadmin-secret` with a secure password for authenticating to ZooKeeper.

```
Server {
    org.apache.zookeeper.server.auth.DigestLoginModule required
        user_zkadmin="zkadmin-secret";
};
```

Make sure the file is not world-readable:

```
chmod 600 $KAFKA_HOME/config/zookeeper_jaas.conf
```

Edit `zookeeper.properties`, adding the following configuration:

```
authProvider.1=\
    org.apache.zookeeper.server.auth.SASLAuthenticationProvider
jaasLoginRenew=3600000
```

The location of the JAAS file can be specified using the `java.security.auth.login.config` system property, passed using the `KAFKA_OPTS` environment variable. (Both Kafka and ZooKeeper use the same wrapper scripts, which work off the same environment variables.) Restart ZooKeeper, but don't connect to it just yet.

```
JAAS_CONFIG=$KAFKA_HOME/config/zookeeper_jaas.conf \
KAFKA_OPTS=-Djava.security.auth.login.config=$JAAS_CONFIG \
$KAFKA_HOME/bin/zookeeper-server-start.sh \
    $KAFKA_HOME/config/zookeeper.properties
```

The next step is to author a client JAAS file that can be used by Kafka as well as CLI tools to connect to ZooKeeper. Create a file named `kafka_server_jaas.conf` in the `$KAFKA_HOME/config` directory, containing the following:

```
Client {
    org.apache.zookeeper.server.auth.DigestLoginModule required
        username="zkadmin"
        password="zkadmin-secret";
};
```

Make sure the file is not group-readable:

```
chmod 600 $KAFKA_HOME/config/kafka_server_jaas.conf
```

After ZooKeeper starts with the server-side JAAS configuration, connections will be authenticated using SASL and clients will be able to set ACLs on znodes as needed. However, any existing znodes that were created *prior* to enabling SASL will remain fully-accessible by the `world` scheme. Recall, ZooKeeper's authentication exists to support its authorization controls. Enabling authentication on its own has no effect on existing znodes. Fortunately, Kafka provides a convenient migration script that edits ACLs on existing znodes, transferring permissions from `world` to an admin user of our choice.

```
JAAS_CONFIG=$KAFKA_HOME/config/kafka_server_jaas.conf \
KAFKA_OPTS=-Djava.security.auth.login.config=$JAAS_CONFIG \
$KAFKA_HOME/bin/zookeeper-security-migration.sh \
    --zookeeper.connect localhost:2181 \
    --zookeeper.acl secure
```

This operation will take a few seconds and should exit without outputting anything to the console.

The migration step is only necessary for existing ZooKeeper ensembles that have previously been initialised by Kafka. Fresh ZooKeeper installations (with no prior data) will function securely without migration, provided that SASL has been enabled prior to connecting Kafka brokers.

Configure the broker to authenticate to ZooKeeper

The next step is to enable ZooKeeper authentication on the broker. Edit `server.properties`, adding the following line:

```
zookeeper.set.acl=true
```

Then, start the broker with the client-side JAAS configuration created in the previous step:

```
JAAS_CONFIG=$KAFKA_HOME/config/kafka_server_jaas.conf \
KAFKA_OPTS=-Djava.security.auth.login.config=$JAAS_CONFIG \
$KAFKA_HOME/bin/kafka-server-start.sh \
    $KAFKA_HOME/config/server.properties
```

Enable ZooKeeper authentication on CLI tools

The reader may have noticed a pattern among the applications that connect to a SASL-enabled ZooKeeper: they all need to set `java.security.auth.login.config` to the location of a JAAS file.

CLI tools are no exception. Once ZooKeeper authentication has been enabled, you must supply a valid JAAS file every time you need to query or modify ZooKeeper's state. The example below shows how the `kafka-config.sh` tool can be parametrised with the location of the JAAS file. Most scripts in `$KAFKA_HOME/bin` support the use of the `KAFKA_OPTS` environment variable for passing arbitrary system properties to the process in question.

```
JAAS_CONFIG=$KAFKA_HOME/config/kafka_server_jaas.conf \
KAFKA_OPTS=-Djava.security.auth.login.config=$JAAS_CONFIG \
$KAFKA_HOME/bin/kafka-configs.sh --zookeeper localhost:2181 \
    --describe --entity-type users
```

The JAAS file may be omitted in the scenario where you might be browsing top-level znodes that are world-readable; for example, using the `zookeeper-shell.sh` tool.

Reverting the configuration

If ZooKeeper authentication ceases to be a requirement, it can be reverted to its original (unauthenticated) state by running `zookeeper-security-migration.sh` with the `--zookeeper.acl unsecure` flag. Afterwards, you may edit `zookeeper.properties` to remove the `authProvider.1` and `jaasLoginRenew` properties. Likewise, the `server.properties` configuration will also need its `zookeeper.set.acl` setting removed or set to `false`.

Note, the rest of the chapter and the book assumes that ZooKeeper authentication is disabled. If you have enabled it out of curiosity, now is a good time to revert the configuration.

Configuring CLI tools

Once Kafka has been configured with SSL and authentication, it isn't just the producer and consumer clients that must be configured; the built-in CLI tools and other administrative applications must also be configured with the location of the client truststore file and the parameters of the authentication flow.

Start by copying the `client.truststore.jks` file (that was generated as part of setting up SSL) to a location where a client can easily access it:

```
cp /tmp/kafka-ssl/client.truststore.jks $KAFKA_HOME/config
```

Next, create a `client.properties` file `$KAFKA_HOME/config`. The contents of the file are listed below. On the author's machine, the truststore is located in `/Users/me/opt/kafka_2.13-2.4.0/config`. You will need to replace this with the absolute path to the truststore file, as most applications don't work well with relative paths. We are going to use the `admin` user for authentication, which is fitting for administrative tools.

```
security.protocol=SASL_SSL
ssl.endpoint.identification.algorithm=https
ssl.truststore.location= \
    /Users/me/opt/kafka_2.13-2.4.0/config/client.truststore.jks
ssl.truststore.password=secret
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config= \
    org.apache.kafka.common.security.scram.ScramLoginModule \
        required \
        username="admin" \
        password="admin-secret";
```

In production deployments you are more likely to have multiple admin-like users with varying permissions. A discussion on user permissions and the broader topic of authorization is deferred until the next section.

Next, run one of the existing CLI tools to verify the configuration. In the example below, we are using `kafka-topics.sh` to list the current topics. The location of the `client.properties` file is supplied via the `--command-config` flag. Note: we are connecting to port `9094` in this example, not `9092`.

```
$KAFKA_HOME/bin/kafka-topics.sh --bootstrap-server localhost:9094 \
    --command-config $KAFKA_HOME/config/client.properties --list
```

The resulting output lists the topics.

```
__consumer_offsets
getting-started
```

Configuring Kafdrop

Having configured the CLI tools, the next step is to ensure that we can connect to a secured cluster using Kafdrop. Like the CLI example, Kafdrop requires a truststore file, as well as a set of properties for configuring SSL and SASL. Unlike the CLI, Kafdrop does not require an absolute path to the truststore file; instead, you can simply leave the truststore file in Kafdrop's root directory.

Start by copying `client.truststore.jks` to the Kafdrop directory, renaming the file to `kafka.truststore.jks` in the process. This is the standard truststore filename that Kafdrop expects by default.

```
cp /tmp/kafka-ssl/client.truststore.jks \
    ~/code/kafdrop/kafka.truststore.jks
```

In addition, create a `kafka.properties` file in Kafdrop's root directory, containing the following:

```
security.protocol=SASL_SSL
ssl.endpoint.identification.algorithm=https
ssl.truststore.password=secret
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config= \
    org.apache.kafka.common.security.scram.ScramLoginModule \
        required \
        username="admin" \
        password="admin-secret";
```

The contents of `kafka.properties` are strikingly similar to the `client.properties` file used in the previous example. There is only one difference: the location of the truststore file is left unspecified. Kafdrop will default to `kafka.truststore.jks` by convention.

Next, start Kafdrop, this time connecting to `localhost:9094`:

```
java -jar target/kafdrop-3.22.0-SNAPSHOT.jar \
    --kafka.brokerConnect=localhost:9094
```

Once connected, Kafdrop will behave identically to previous examples. The only difference now is that it will connect over SSL and will authenticate itself using the `admin` user. This is important for our upcoming examples, which use Kafka's authorization capabilities. Unless Kafdrop is set up to use SASL and SSL, it will be of no use once the Kafka cluster has been fully hardened. The same can be said about built-in CLI tools and any third-party tools one might be using.

Authorization

With SSL and authentication ticked off, we are two-thirds of the way there. Authentication is necessary to identify the user principal, and that is immensely useful for allowing or blocking access to individual clients. Quite often, especially when the cluster is shared across multiple applications, the binary allow-or-deny option is insufficient; we need more fine-grained control over the actions that an authenticated client should be allowed to perform. This is where *authorization* comes into the picture.

Kafka implements authorization by way of resource-centric ACLs. An ACL specifies the following:

- **Principal** — the user performing the operation.
- **Operation** — the action being performed. For example, `Read` or `Write`.
- **Host** — an optional restriction on the addresses that the rule applies to.
- **Resource type** — the type of subject on which the action is to be performed. For example, `Topic` or `Group`.
- **Resource pattern** — a simple rule for matching the subject by its name. Rules specify a single resource by its literal name, or can group multiple resources using prefixes or wildcards.
- **Outcome** — whether the action should be allowed or denied.

Both Kafka and ZooKeeper utilise ACLs for authorization; however, their implementations are distinct and should not be confused. The table below outlines the differences.

ZooKeeper ACLs	Kafka ACLs
Apply to discrete resources.	Can specify multiple resources using prefixes and wildcards.
Restrict operations to users.	Restrict operations to users or network addresses.
Can only allow an operation.	Can allow or deny operations.

There are several supported operations:

- `Read` — view a data-centric resource (for example, a topic), but not write to its contents.
- `Write` — modify the contents of a data-centric resource.
- `Create` — create a new resource.
- `Delete` — delete a resource.
- `Alter` — change the contents of a non-data resource (for example, ACLs).
- `Describe` — get information about a resource.
- `ClusterAction` — operate on the cluster. For example, fetch metadata or initiate a controlled shutdown of a broker.
- `DescribeConfigs` — retrieve the configuration of a resource.
- `AlterConfigs` — alter the configuration of a resource.
- `IdempotentWrite` — perform an idempotent write. Idempotent writes are described in Chapter 10: Client Configuration.
- `All` — all of the above.

The operations act on resource types. The following is a list of supported resource types and related operations:

- `Cluster` — the entire Kafka cluster. Supports: `Alter`, `AlterConfigs`, `ClusterAction`, `Create`, `Describe` and `DescribeConfig`.
- `DelegationToken` — a delegation token. This resource type is different from the others in that it is bound by special rules, where permissions apply to the owner and renewer. Supports: `Describe`.
- `Group` — a consumer group. Supports: `Delete`, `Describe` and `Read`.
- `Topic` — a topic. Supports: `Alter`, `AlterConfigs`, `Create`, `Delete`, `Describe`, `DescribeConfigs`, `Read` and `Write`.
- `TransactionalId` — transactional sessions of the same logical producer. Supports: `Describe` and `Write`.

The lists above may appear confusing, especially the combination of resource types and operations. Kafka's official documentation provides a more detailed matrix of the relationships between low-level protocol messages (or API calls), operations and resource types — not that it makes things a whole lot clearer. (Not unless you are intimately familiar with the underlying protocol messages.) This might not sound very encouraging, but most ACL-related issues are diagnosed and resolved through experimentation and consulting people who have seen a similar problem before. Admittedly, it's not great, but it's the best answer there is.

To make this process less ambiguous, Kafka clients will throw specific subclasses of an `AuthorizationException` if the evaluation of rules results in a *deny* outcome for one of the five resources types. For example, a deny against a topic will result in a `TopicAuthorizationException`, a deny against a group will lead to a `GroupAuthorizationException`, and so forth.

There will be times where a developer will swear they have enabled an operation on a resource, such as a topic, only to get some obscure `ClusterAuthorizationException` at runtime. It might seem nonsensical at the time — one is trying to publish a record, but is inexplicably being knocked back at the cluster level. Confusions such as these typically occur when a client action entails multiple API calls, and one of those calls is being denied.

Enable authorization on the broker

To enable authorization, edit `server.properties`, adding the following lines:

```
authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer
super.users=User:admin
```

Restart the broker for changes to take effect.

The `super.users` property specifies a semicolon-delimited list of privileged users that are able to perform *any* operation. In this example, we specified `admin` for convenience, given that we already have this user configured, and we are also signing on with the same user for interbroker authentication. You may specify other users, but make sure the interbroker user remains in the list.

Kafka delimits usernames with semicolons because a comma is a valid character in a username that has been derived from the attributes of an SSL certificate.

As it is generally accepted, operating a secure cluster implies that both client and interbroker operations are subject to authentication and authorization controls. The interbroker operations are split into two broad classes: *cluster* and *topic*. Cluster operations refer to the actions necessary for the management of the cluster, such as updating broker and partition metadata, changing the leader and the in-sync replicas of a partition, and initiating a controlled shutdown. One of the defining responsibilities of a broker is the replication of partition data. Because of the way replication

works internally, it is also necessary to grant topic access to brokers. Specifically, replicas must be authorized for both `Read` and `Describe` operations on the topics being replicated. The `Describe` permission is granted implicitly, as long as the `Read` permission is present.

> Kafka takes a *default-deny* stance, which is otherwise known as a *positive* or *additive* security model. Unless an attempted action is explicitly allowed, it is assumed to be denied. This behaviour can be inverted by adding the following value to `server.properties`:
>
> ```
> allow.everyone.if.no.acl.found=true
> ```
>
> Generally speaking, the default-deny model is recommended over the default-allow, as it takes a more conservative approach to dispensing access. One should think of it as an additive approach: starting from zero — a blank slate, adding permissions on a needs basis. At any point, it can be clearly reasoned as to who has access to a resource — by simply looking over the permissions list. Unless an *allow* rule explicitly stating the *principal* and the *resource* in question is present, then there can be no access-granting relationship between the pair of entities.

Once the broker restarts, we can use the `kafka-acls.sh` tool to view and manipulate ACLs. The command below will list all configured ACLs, internally utilising the Kafka Admin API.

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --list
```

It should come up with an empty list of rules. This is to be expected as we haven't yet configured any.

When specifying a bootstrap list via the `--bootstrap-server` flag, it is also necessary to state the location of the JAAS file used for authenticating to Kafka. When Kafka is running with authorization enabled, users cannot just view and alter ACLs — not unless they have been explicitly authorized to do so. By connecting with the `admin` user, we are effectively operating in 'god mode'.

> An alternate way of viewing and setting ACLs is to connect to ZooKeeper directly. This will work provided ZooKeeper is reachable from the network where the command is run, and ZooKeeper is not running with authentication and ACLs of its own. The command below shows how `kafka-acls.sh` can be used directly with an unauthenticated ZooKeeper node.
>
> ```
> $KAFKA_HOME/bin/kafka-acls.sh \
>     --authorizer-properties zookeeper.connect=localhost:2181 \
> ```

```
        --list
```

At this point, we will have a functioning broker with no ACLs. This means that, with the exception of the `admin` user, no other client will be able to interact with Kafka. Of course, the point of ACLs is not to lock out all users, but restrict access to those users who legitimately need to view or manipulate specific resources in the cluster. Previously, in the SASL examples, the user `alice` published to the `getting-started` topic. Running this example now results in a series of warnings and errors:

```
21:55:56/1854  WARN  [kafka-producer-network-thread | producer-1]: ␣
    [Producer clientId=producer-1] Error while fetching metadata ␣
    with correlation id 4 : ␣
    {getting-started=TOPIC_AUTHORIZATION_FAILED}
21:55:56/1854  ERROR [kafka-producer-network-thread | producer-1]: ␣
    [Producer clientId=producer-1] Topic authorization failed for ␣
    topics [getting-started]
Published with metadata: null, error: org.apache.kafka.common. ␣
    errors.TopicAuthorizationException: Not authorized to access ␣
    topics: [getting-started]
```

This particular problem can be fixed by adding read and write permissions to user `alice`:

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:alice \
    --operation Read --operation Write --topic getting-started
```

Which results in the following output:

```
Adding ACLs for resource `ResourcePattern(resourceType=TOPIC, ␣
      name=getting-started, patternType=LITERAL)`:
  (principal=User:alice, host=*, operation=WRITE, ␣
      permissionType=ALLOW)
  (principal=User:alice, host=*, operation=READ, ␣
      permissionType=ALLOW)

Current ACLs for resource `Topic:LITERAL:getting-started`:
  User:alice has Allow permission for operations: ␣
      Read from hosts: *
  User:alice has Allow permission for operations: ␣
      Write from hosts: *
```

However, this is not sufficient. As it was previously stated, some operations require several API calls. In our producer example, we have set the property `enable.idempotence` to `true` Enabling idempotence ensures that the producer maintains strict order and does not write duplicates in the event of an intermittent network error. However, this property requires cluster-level permission for the `IdempotentWrite` operation:

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:alice \
    --operation IdempotentWrite --cluster
```

Resulting in:

```
Adding ACLs for resource `ResourcePattern(resourceType=CLUSTER, ␣
      name=kafka-cluster, patternType=LITERAL)`:
  (principal=User:alice, host=*, operation=IDEMPOTENT_WRITE, ␣
      permissionType=ALLOW)

Current ACLs for resource `Cluster:LITERAL:kafka-cluster`:
  User:alice has Allow permission for operations: ␣
      IdempotentWrite from hosts: *
```

Running the `kafka-acls.sh` command with the `--list` switch now results in a complete ACL listing, showing `alice` and her associated permissions:

```
Current ACLs for resource `Cluster:LITERAL:kafka-cluster`:
  User:alice has Allow permission for operations: ␣
      IdempotentWrite from hosts: *

Current ACLs for resource `Topic:LITERAL:getting-started`:
  User:alice has Allow permission for operations: ␣
      Read from hosts: *
  User:alice has Allow permission for operations: ␣
      Write from hosts: *
```

Try publishing now. It should work like a charm.

Now, let's switch back to the consumer sample that we ran as part of setting up SASL. That example authenticated as `alice` and read from the `getting-started` topic using the `basic-consumer-sample` consumer group. Running this example now results in the following error:

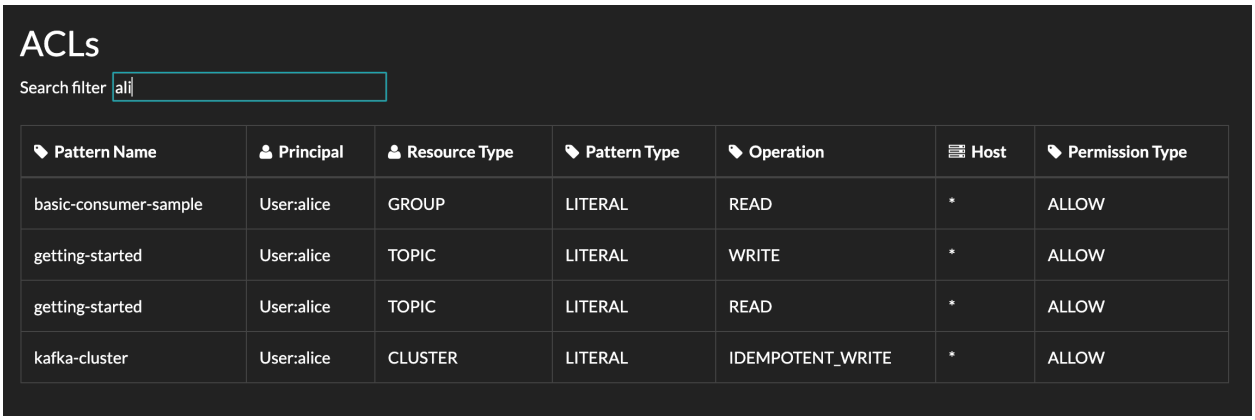
```
Exception in thread "main" org.apache.kafka.common.errors. ⮒
    GroupAuthorizationException: Not authorized to access ⮒
    group: basic-consumer-sample
```

The issue is that the ACLs for topics are distinct to the ACLs for consumer groups, as the two are treated as distinct resources. To fix the issue above, add the `Read` permission on the consumer group:

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:alice \
    --operation Read --group basic-consumer-sample
```

The consumer should now be able to function normally. And there we have it: the world-readable `getting-started` has just been converted into a *high-assurance* topic.

ACLs can be viewed broadly for all users and resources using the Kafdrop tool. It also provides a convenient filter text box, letting you search for specific principals or resource patterns. The screenshot below shows how the ACLs we have defined so far appear in Kafdrop.



Kafdrop — ACls for the user 'alice'

> In our examples, we have used `alice` to demonstrate both read (consume) and write (publish) privileges. As a convenience, this was sufficient. However, in practice, it is rare to see both the producer and the consumer acting as one entity for the same topic. More often, they are distinct. It is considered best-practice to assign individual usernames to application entities. Furthermore, an entity should only be granted the minimal set of privileges that it legitimately requires to fulfil its responsibilities.
>
> In the same vein, the correct approach with respect to admin users is to avoid them. It is best to create a dedicated user for interbroker communications, having the necessary level of cluster and

topic access, but no ability to create new users or modify ACLs.

Removing permissions

To remove a permission, simply run `kafka-acls.sh` with the `--remove` switch in place of `--add`, with the rest of the flags used for adding a permission, as shown in the example below.

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --authorizer-properties zookeeper.connect=localhost:2181 \
    --remove --allow-principal User:alice \
    --operation Read --operation Write --topic getting-started \
    --force
```

The use of the `--force` switch prevents the operator from having to respond to a *yes/no* safety prompt.

Later in the chapter, we will look at bulk-removing permissions.

Mixing allow and deny permissions

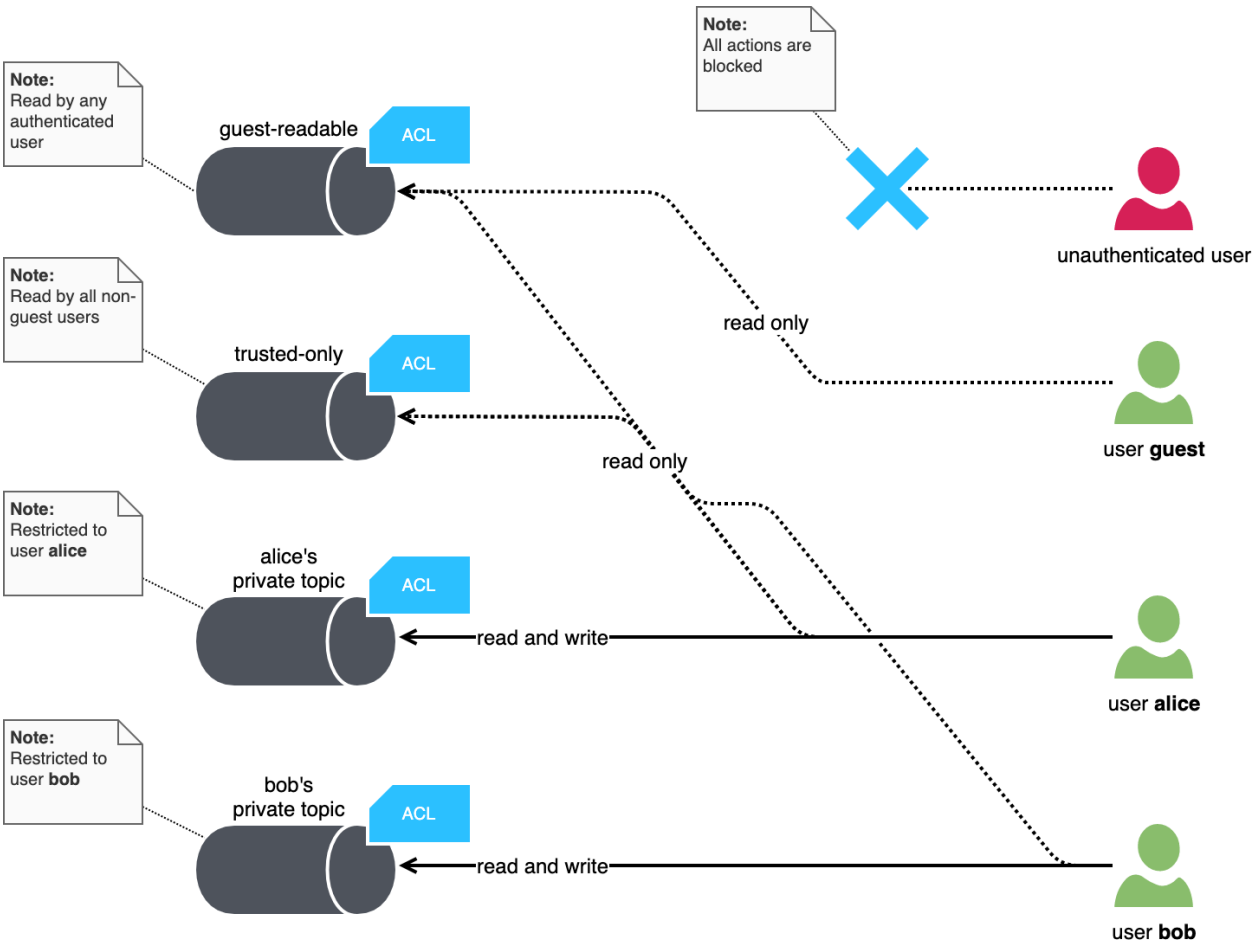
One of the known complications of Kafka's rules is that they support both allow and deny actions. This makes them more flexible, but also easier to misconfigure. For example, you can allow access to all users on a specific topic, but then disallow access to a smaller group of users. A natural response might be: *Doesn't this contradict the default-deny model, and if not, where would this capability be useful?*

One way of looking at it, is that it allows us to interleave default-allow and default-deny tactics in the same overall security model, creating a hierarchy of sorts. To understand where this might be beneficial, consider the following challenge, involving a hypothetical Kafka cluster that permits guest level access for a small subset of resources. Guest credentials can be procured relatively easily — there might even be a pre-canned set of shared credentials that are freely available. Assuming strong network-level controls are in place, guest access is only permitted from a trusted network.

We might be publishing something trivial on a topic which we would like to make available to guests for reading, as well as to trusted users. Let's call this *low-assurance* topic `guest-readable`. Now, assume there is another topic which contains information for authenticated users that a have higher level of clearance than `guest` — a *medium assurance* topic. We will call it `trusted-only`. And finally, the cluster may contain numerous *high-assurance* topics that are admissible to specific

users. Frankly, the high assurance topics are least interesting, as they can be trivially fortified using standard ACLs. They were only added to the challenge for completeness, to make it more life-like.

To make this challenge more interesting, we do not know in advance who all the users are. All that can be stated with certainty is the username of the guest user — `guest`. Any other user that is authenticated, but not `guest`, is a trusted user and should have read access to the `trusted-only` topic. The diagram below illustrates the intended target state for this challenge, depicted the relationships between users and topics.



The ACL challenge

Normally, we would add *allow* rules to the topics, until all access requirements are satisfied. The complication here is that we don't have a definitive list of usernames to work with; furthermore, even if we did, the addition of new users would require updates to a vast number of ACLs, making their maintenance unsustainable. Ideally, this problem is solved with Role-Based Access Control (RBAC); however, the out-of-the-box Kafka setup does not support this.

There are third-party extensions to Kafka that support RBAC. This book is predominantly focused on the foundational insights and advanced skills — featuring a vanilla Kafka setup, as well as the minimally essential set of third-party tools. RBAC is on the far-right of the notional 'extensibility' spectrum, and has not been considered for inclusion into the base Kafka offering. As such, we will not explore it further.

Before solving this challenge, we need to create our fixtures. We need a pair of topics — `guest-readable` and `trusted-only`, and we need the `guest` user. Let's get started by creating these entities:

```
# add a user (requires ZooKeeper access)
$KAFKA_HOME/bin/kafka-configs.sh --zookeeper localhost:2181 \
    --alter \
    --add-config 'SCRAM-SHA-512=[password=guest-secret]' \
    --entity-type users --entity-name guest


# create the guest-readable topic
$KAFKA_HOME/bin/kafka-topics.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --create --topic guest-readable \
    --partitions 1 --replication-factor 1


# create the trusted-only topic
$KAFKA_HOME/bin/kafka-topics.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --create --topic trusted-only \
    --partitions 1 --replication-factor 1
```

The `kafka-topics.sh` CLI supports the creation of topics via a bootstrap list, provided you give it the appropriate credentials and truststore. On the other hand, the `kafka-configs.sh` tool is limited in the entity types that can be configured via a broker. Entities such as users must be configured directly in ZooKeeper.

The next step is to try reading from the `guest-readable` topic using the `guest` user. Rather than using our existing SASL consumer example, we will change the code slightly to run it using a free consumer — in other words, without an encompassing consumer group. This particular variation replaces the `Consumer.subscribe()` call with `Consumer.assign()`, removing the need to pass in the `group.id` consumer property. The complete listing is shown below.

```
import static java.lang.System.*;

import java.time.*;
import java.util.*;
import java.util.stream.*;

import org.apache.kafka.clients.*;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.*;
import org.apache.kafka.common.config.*;
import org.apache.kafka.common.security.scram.*;
import org.apache.kafka.common.serialization.*;

public final class SaslSslFreeConsumerSample {
  public static void main(String[] args) {
    final var topic = "guest-readable";

    final var loginModuleClass = ScramLoginModule.class.getName();
    final var saslJaasConfig = loginModuleClass
        + " required\n"
        + "username=\"guest\"\n"
        + "password=\"guest-secret\";";

    final var config = new HashMap<String, Object>();
    config.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9094");
    config.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG,
               "SASL_SSL");
    config
    .put(SslConfigs.SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG,
         "https");
    config.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,
               "client.truststore.jks");
    config.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG, "secret");
    config.put(SaslConfigs.SASL_MECHANISM, "SCRAM-SHA-512");
    config.put(SaslConfigs.SASL_JAAS_CONFIG, saslJaasConfig);
    config.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
               StringDeserializer.class.getName());
    config.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
               StringDeserializer.class.getName());
    config.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
    config.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
```

```
    try (var consumer = new KafkaConsumer<String, String>(config)) {
      final var partitionInfos = consumer.partitionsFor(topic);
      final var topicPartitions = partitionInfos.stream()
          .map(partInfo -> new TopicPartition(partInfo.topic(),
                                              partInfo.partition()))
          .collect(Collectors.toSet());
      consumer.assign(topicPartitions);

      while (true) {
        final var records = consumer.poll(Duration.ofMillis(100));
        for (var record : records) {
          out.format("Got record with value %s%n", record.value());
        }
      }
    }
  }
}
```

Running this code will produce the following error:

```
Exception in thread "main" org.apache.kafka.common.errors. ↪
    TopicAuthorizationException: Not authorized to access ↪
    topics: [guest-readable]
```

And indeed, that is to be expected — after all, we had just created the `guest` user. We need to grant the due permissions to the `guest` user. But here's the snag: if we keep granting permissions additively, then we have to specify all users. And as it was stated in the challenge, the usernames are now known in advance.

Rather than allowing `guest-readable` for all known users individually, we simply make it world-readable. Here 'world' is a bit of a misnomer — we aren't actually exposing the topic to the world — only to those users that have successfully authenticated via the `SASL_SSL` listener. Run the command below.

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:"*" \
    --operation Read --topic guest-readable
```

Compared to the earlier examples, we are using a wildcard (quoted asterisk character) in place of a username. The wildcard can be specified using single or double-quotes. Avoid using the wildcard

without quotes, as most shells will expand the * character to a list of files in the current directory of the caller.

To make the testing easier, we should also publish some records to the `guest-readable` and `trusted-only` topics. This can be accomplished using the `kafka-console-producer.sh` CLI:

```
$KAFKA_HOME/bin/kafka-console-producer.sh \
    --producer.config $KAFKA_HOME/config/client.properties \
    --broker-list localhost:9094 \
    --topic guest-readable
```

Key in a few records, and press `CTRL-D` when done. Repeat for the `trusted-only` topic.

Run our consumer code again. This time it should work. You can also change the user from `guest` to `alice`, and it will still work as expected.

With the low-assurance topic out of the way, the next part of the challenge is to make our medium-assurance `trusted-only` topic behave as intended. The solution, as the reader would have guessed, is to mix allow and deny outcomes.

Start by making the `trusted-only` topic world-readable:

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:"*" \
    --operation Read --topic trusted-only
```

Then, follow up by excluding the `guest` user from the `trusted-only` topic, using the `--deny-principal` flag:

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --deny-principal User:guest \
    --operation Read --topic trusted-only
```

Essentially, the above is saying: *"make 'trusted-only' readable by everyone except the 'guest' user"*, which is exactly what is needed.

Run the code again, switching the user to `guest` and the topic to `trusted-only`. We should see a `TopicAuthorizationException`. However, if we switch the user to `alice`, the topic's contents will be revealed.

Deny rules take precedence over *allow* rules, irrespective of the granularity of the resource patterns identified by those rules. Therefore, you should always use an *allow* rule over a broader-matching

resource pattern than a *deny* rule. Stated otherwise, deny a subset of the resources that you had previously allowed, not the other way round. Had we attempted to reverse the rules of the challenge — deny everyone but allow a specific user, it would not have worked — the *deny* rule would have had the final say. In practice, when combining *allow* and *deny* outcomes, the outer-most *deny* rule is implicit — enforced by Kafka's own *default-deny* policy. This gives us the flexibility of (up to) three rule tiers — the outer-most *default-deny*, followed by a custom *allow* rule, and finally by a custom *deny* rule.

Literal and prefixed resource patterns

Most ACL examples so far have involved the literal matching of resources and user principals. In other words, we specified the *exact* name of the topic and the *exact* username, along with a concrete allow/deny outcome. These are called *literal* matches. In one example, when setting up world-readable topics, we resorted to the use of the `User:"*"` *wildcard* match on the user principal.

It should be noted that wildcards in Kafka ACLs behave very differently to the familiar wildcards used to match files and directories in the UNIX and Windows command shells. When used on its own, a wildcard implies *"match all entities"*. However, a wildcard cannot be combined with literal text to mean *"match a part of a string"*.

Kafka resolved the issue of partial resource matching in KIP-290[41] as part of release 2.0.0 — with the introduction of prefixed ACLs, addressing some of the long-standing limitations of its multitenancy capabilities.

The default pattern type is *literal*, which is the equivalent of invoking `kafka-acls.sh` with the `--resource-pattern-type=literal` flag. Partial matches can be accomplished using the `--resource-pattern-type=p`
For example, the following denies `guest` from all topics that begin with the literal string `trusted`:

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --deny-principal User:guest \
    --resource-pattern-type=prefixed \
    --operation Read --topic trusted
```

Literal strings and wildcards can be used to identify resources, as well as to identify user principals. Prefixed matching applies only to resources — it is not possible to prefix-match a user.

[41] https://cwiki.apache.org/confluence/x/QpvLB

Listing and bulk-removal of permissions using the CLI

It has been shown how rules can be created and removed using the CLI. The most convenient way to list rules, provided you are working in a browser environment, is via Kafdrop's ACL screen. In those cases where Kafdrop is not at hand, the `kafka-acls.sh` can be used to query rules and even remove them in bulk.

We've already seen how the `--list` switch can be used to display the rules. But a production environment containing hundreds or thousands of ACL entries makes the unfiltered `--list` command unwieldy. Running `--list` (with no other predicates) on the state of the ACLs accumulated from all previous examples results in the following:

```
Current ACLs for resource `Topic:LITERAL:guest-readable`:
  User:* has Allow permission for operations: Read from hosts: *

Current ACLs for resource `Cluster:LITERAL:kafka-cluster`:
  User:alice has Allow permission for operations: □
      IdempotentWrite from hosts: *

Current ACLs for resource `Topic:PREFIXED:trusted`:
  User:guest has Deny permission for operations: Read from hosts: *

Current ACLs for resource `Topic:LITERAL:getting-started`:
  User:alice has Allow permission for operations: Read from hosts: *
  User:alice has Allow permission for operations: Write from hosts: *

Current ACLs for resource `Topic:LITERAL:trusted-only`:
  User:guest has Deny permission for operations: Read from hosts: *
  User:* has Allow permission for operations: Read from hosts: *

Current ACLs for resource `Group:LITERAL:basic-consumer-sample`:
  User:alice has Allow permission for operations: Read from hosts: *
```

The output is large but still useful. In fact, we can observe that the `Deny` permission for the literal `trusted-only` topic is redundant, as it is covered by the broader-matching `Deny` permission on the prefix `trusted` topic; both targeting the same user principal `alice`.

The basic `--list` switch can be elaborated upon using two additional resource pattern types: `any` and `match`. The `any` pattern type locates all rules that match the specified resource name *exactly*, which includes literal, wildcard and prefixed patterns.

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --list --resource-pattern-type=any \
    --topic trusted-only
```

This command results in the following:

```
Current ACLs for resource `ResourcePattern(resourceType=TOPIC, □
    name=trusted-only, patternType=LITERAL)`:
  (principal=User:*, host=*, operation=READ, permissionType=ALLOW)
  (principal=User:guest, host=*, operation=READ, permissionType=DENY)
```

The `any` pattern type has located two rules of the three that apply to the `trusted-only` topic. It didn't match the prefixed pattern for the name `trusted`, even though the latter also applies to the `trusted-only` topic. To include all rules that affect a given resource, change the pattern type type to `match`. The result:

```
Current ACLs for resource `ResourcePattern(resourceType=TOPIC, □
    name=trusted-only, patternType=LITERAL)`:
  (principal=User:*, host=*, operation=READ, permissionType=ALLOW)
  (principal=User:guest, host=*, operation=READ, permissionType=DENY)

Current ACLs for resource `ResourcePattern(resourceType=TOPIC, □
    name=trusted, patternType=PREFIXED)`:
  (principal=User:guest, host=*, operation=READ, permissionType=DENY)
```

Splendid. All three rules have been located. The `match` pattern type is convenient for debugging ACLs issues and reasoning about the access rights to a specific resource. Using a `match` query effectively lets us take a look at ACLs with the eyes of Kafka's built-in `SimpleAclAuthorizer`.

The ability to round up several rules for a resource is also convenient when performing a bulk delete operation. To remote the matching rules, run the commands above using the `--remove` switch in place of `--list`. Bear in mind that the remove command does not preview the rules before deleting them, although it does provide for a *yes/no* safety prompt, unless executed with the `--force` switch. Consequently, always run the `--list` command to ascertain the complete list of affected rules before removing them.

> In the examples above, we first created the `guest` user, then the two topics, then associated the user with the topics through several ACL rules. Kafka allows us to carry out these steps in any order. When creating a rule, there is no requirement that either the principal or the resource must exist. There is no referential dependency relationship between these three entities. As a consequence, it is also possible to delete users and topics despite having ACL rules that reference them.

As much as it is a convenience, it is also a 'gotcha': it is possible to accidentally create a rule that references a mistyped user or topic — provided the rule is well-formed, Kafka will vacuously persist it. When creating ACL rules, particularly when these rules relate to high-assurance resources, it is considered best-practice to validate these rules. The best way to do this is programmatically, by writing a script that performs specific actions and asserts the allow or deny outcomes. However, the cost of getting the test wrong may be catastrophic — imagine asserting a `Delete` operation that succeeds when it is expected to fail. The result is the deletion of resources — not just a failure of the test, but the loss of potentially vital data and the disruption of downstream consumers.

The recommended way to test rules is to operate two clusters — a *staging* and a *production* setup — where the production ACLs are only manipulated via pre-canned scripts that have been validated against the staging cluster. This strategy implies that users are mirrored in both clusters and have identical permissions. In that sense, a staging cluster is quite different from a typical non-production cluster, where users might routinely be given elevated permissions to enable productivity.

Network address restrictions

In addition to restrictions on user principals, Kafka allows rules to specify the network addresses of connected clients. This is done with the `--allow-host` or `--deny-host` flags, with their sole argument being the target IP address. Multiple `--allow-host` and `--deny-host` flags may be specified in a single command, resulting in the addition of multiple rules.

The hosts passed to `--allow-host` and `--deny-host` must be specific addresses — in either IPv4 or IPv6 form. Hostnames and IP address ranges are not presently supported. KIP-252[42] proposes to extend the existing ACL functionality to support address ranges as well as groups of addresses in CIDR notation; however, this KIP is still in its infancy.

While the option is present, the changing landscape of application architecture — the proliferation of Cloud-based technologies, containerisation, NAT, and elastic deployment topologies, make it difficult to recommend host-based rules as a legitimate form of application-level access control. It is becoming more challenging and sometimes downright intractable to isolate application processes based on their network address. Where this makes sense, controlling access based on origin addresses is best accomplished with a firewall — a device that is designed for this very purpose and provides effective network and transport-layer controls. Furthermore, firewalls understand ports, protocols and, crucially, address ranges. The latter represents an absolute requirement when dealing with elastic deployments. At the application layer, access should be a function of user principals and, ideally, their roles.

[42] https://cwiki.apache.org/confluence/x/jB6HB

Common authorization scenarios

Although there are numerous supported operations and resource types supported by Kafka's authorization model, the majority of authorization needs fall under a handful of use cases. These are presented here.

Creating topics

The creation of a topic requires `Create` privileges on the `Topic` resource type. The topic name can be literal, although more often topic creation and publishing rights are granted over a prefixed resource pattern. In the example below, user `bob` is granted permission to create any topic that begins with the string `prices.`.

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:bob \
    --resource-pattern-type=prefixed \
    --operation Create --topic prices.
```

Deleting topics

Topic deletion privileges are the logical opposite of creation, replacing the `Create` operate with `Delete`:

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:bob \
    --resource-pattern-type=prefixed \
    --operation Delete --topic prices.
```

Publishing to a topic

Like creating a topic, publishing is often granted as a prefixed permission — requiring `Write` capability on the `Topic` resource type.

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:bob \
    --resource-pattern-type=prefixed \
    --operation Write --topic prices.
```

If the publisher requires idempotence enabled, then a further `IdempotentWrite` operation must be allowed for the `Cluster` resource type:

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:bob \
    --operation IdempotentWrite --cluster
```

Consuming from a topic

For a free consumer to read from a topic, it requires the `Read` operation on the `Topic` resource type. Although consumer permissions may be specified using prefixed patterns, the more common approach is to issue literal credentials to minimise unnecessary topic exposure — in line with the *Principle of Least Privilege*. To grant `bob` read rights to the `prices.USD` topic, issue the following command.

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:bob \
    --operation Read --topic prices.USD
```

If the consumer operates within the confines of a consumer group, then the `Read` operation on the `Group` resource type is required. Often, the consumer group is named after the user because that's how the user-group relationship tends to unfold in practice. A consumer group is effectively a private load-balancer, and in the overwhelming majority of cases, the partition load is distributed across instances of the same application. So it's safe to assume that users will want to create their own consumer groups. This can be accomplished using a prefix-based convention. To grant `bob` rights to all consumer groups beginning with `bob.`, run the following command.

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:bob \
    --resource-pattern-type=prefixed \
    --operation Read --group bob.
```

The facets of information security are numerous and varied, and it can be said without exaggeration that securing Kafka is an epic journey. If there is one thing to be taken away from this chapter, it is that threats are not confined to external, anonymous actors. Threats can exist internally and may persist for extended periods of time, masquerading as legitimate users and occasionally posed by them.

The correct approach to securing Kafka is a layered one. Starting with network policy, the objective is to logically segregate networks and restrict access to authorized network segments, using virtual networking if necessary to securely attach edge networks to operational sites. Network policies block clients operating out of untrusted networks and erect barriers to thwart internal threads, restricting their ability to access vital infrastructure components.

Having segregated the network, the attention should be turned to assuring the confidentiality and integrity of communications. This is accomplished using transport layer security, which in Kafka is implemented under the SSL moniker. The use of X.509 certificates complements SSL, providing further assurance to the clients as to the identity of the brokers — verifying that the brokers are who they claim to be. SSL can be applied both to client-to-broker and interbroker communications. We also have learned of the present limitations in Kafka *vis à vis* SSL — namely, the inability to use SSL to secure broker-to-ZooKeeper communications.

SSL can also be used in reverse, acting as an authentication mechanism — attesting the identity of the connected client to the broker. On the point of authentication, Kafka offers a myriad of options using SASL. These range from the enterprise-focused GSSAPI (Kerberos), to PLAIN and SCRAM schemes for username/password-based authentication, OAuth 2.0 bearer tokens, and finally to delegation tokens that simplify the authentication process for large hordes of ephemeral worker nodes. Authentication isn't just limited to broker nodes; it can also be enabled on ZooKeeper — limiting one's ability to view and modify the contents of potentially sensitive znodes.

Authentication, as we have learned, is really a stepping stone towards authorization. There is little benefit in knowing the identity of a connected party if all parties are to be treated equally. Parties — being clients and peer brokers — bear different roles within the overall architecture landscape. They have different needs and are associated with varying levels of trust. Kafka combines elaborate, rule-based ACLs with a default-deny permission model to create a tailored access profile for every client.

The layered security controls supported by Kafka, in conjunction with externally-sourced controls — such as firewalls, VPNs, transparent TLS proxies and storage encryption — collectively form an

environment that is impregnable against a broad range of threat actors. They enable legitimate actors to operate confidently in a secure environment.

Chapter 17: Quotas

There has been a strong emphasis throughout this book on Kafka's role as a proverbial 'glue' that binds disparate systems. At the very core of the event streaming paradigm is the notion of multiple tenancies.

Chapter 16: Security has set the foundation for multitenancy — delineating how clients having different roles and objectives can securely connect to, and share broker infrastructure — the underlying topics, consumer groups and other resource types. Despite the unmistakable overlap between security and quotas, the latter stands on its own. The discussion on quotas transcends security, affecting areas such as *quality of service* and *capacity planning*. In saying that, the use of quotas requires authentication controls; therefore, Chapter 16: Security is a prerequisite for this chapter.

The examples in this chapter will not work unless authentication has been enabled on the broker and suitable client-side preparations have been made. If you have not yet read Chapter 16: Security and worked through the examples, please do so before proceeding with the material below.

The rationale behind quotas

Mitigating denial of service attacks

As it has been just said, the discussion on quotas is a logical extension of the 'security' topic. At the heart of information security are three primordial concepts, often referred to as the *CIA triad*. (Not to be confused with the intelligence agency.) The acronym deciphers to *confidentiality*, *integrity* and *availability* — phrased in relation to information assets. Chapter 16: Security touched on all aspects of the CIA triad, but focused mostly on confidentiality and integrity, using specific security controls such as encryption (TLS), X.509 certificates, authentication (Kerberos, SASL, OAuth) and authorization (ACLs). The availability aspect is partly catered to by the authorization control: by ensuring that parties are only acting in a manner that has been prescribed for them, we can protect other parties from unsanctioned interference.

The flexibility of Kafka's rule-based ACLs enables us to apply varying levels of assurance to different resources. We might have low-assurance topics collocated with high-assurance topics, where a greater number of semi-trusted clients may be allowed to access the former, admitting a much more select group of clients to the latter. In another scenario, we might be operating a SaaS business, where resources are partitioned on a per-customer basis and where it is essential that customers cannot access or manipulate each other's data.

Even with fine-grained ACLs in place, an authorized client with the lowest level of access may attempt to monopolise cluster resources with the intent of disrupting the operation of legitimate clients. This might be a client with read-only access to some innocuous topic. Alternatively, in the SaaS scenario, it may be a low-tier paying customer whose sole intention is to saturate the service provider and thereby cause financial harm.

The mechanism for a *denial of service* (DoS) attack is fairly straightforward. Once a client gains access to a resource, it can generate large volumes of read queries or write requests (depending on its level of access), thus causing network congestion, memory pressure and I/O load on the brokers. As these are all finite resources, their disproportionate consumption by one client creates starvation for the rest.

Quotas fill the gap left by ACLs, specifying the extent of resource utilisation that is to be accorded to a user. Where that limit has been breached, the brokers will automatically activate mitigating controls, throttling a client until its request profile complies with the set quota. There are no further penalties applied to the offending client beyond throttling. This is intended, as aside from a traffic spike, there is nothing to suggest that a client is malicious; it may simply be responding to elevated levels of demand.

Capacity planning

With multiple clients contending for the use of a common Kafka cluster, how can the operator be sure that the finite resources available to the cluster are sufficient to meet the needs of its clients?

This answer leads to the broader topic of capacity planning. This is a complex, multi-disciplined topic that includes elements of measurement, modelling, forecasting and optimisation. And it is fair to say that this material is well outside our scope. However, the first step of capacity planning is modelling the demand, and quotas are remarkably helpful in this regard. If all current and prospective users of a cluster have been identified, and each has had their quotas negotiated, then the aggregate 'peak' demand can be determined. Whether or not the cluster's resources (disks, CPU, memory, network bandwidth, etc.) will be provisioned to cover the worst-case demand is a separate matter — the balance of cost and willingness to accept the risk of failing to meet demand.

Quality of service

Quality of service (QoS) is strongly related to both the security and capacity aspects. QoS focuses on the customer (or the client, in the context of Kafka), ensuring that the latter receives a service that meets or exceeds the baseline warranted by the service provider. This measure is, in simple terms, a function of the provider's ability to furnish sufficient capacity when it is called for, as well as its resistance to DoS attacks.

QoS naturally dovetails into operational-level agreements (OLAs) — arrangements between collaborating parties that influence the consuming party's ability to provide a service to its downstream consumer — ultimately affecting support-level agreements (SLAs) at the organisation's boundary.

Without specific QoS guarantees, OLAs cannot be reliably fulfilled — that is, in the absence of excessive over-provisioning of resources. The latter is expensive and wasteful; in most cases, it is more economically viable to ration resources than to the purchase excess capacity, unless the cost of rationing exceeds the cost of the resources that are being preserved.

When operating a small cluster with only a handful of connected clients, the baseline capacity is often already in excess of the peak demand, particularly when the cluster comprises multiple brokers for availability and durability. In such deployments, Kafka's efficiency provides for ample headroom, and managing quotas may not be the most productive use an operator's time and resources. As the number of clients grows and their diversity broadens, the need to manage quotas becomes more apparent.

Types of quotas

Kafka supports two types of quotas:

1. **Network bandwidth quotas** — inhibit producers and consumers from transferring data above a set rate, measured in bytes per second.
2. **Request rate quotas** — limit a client's CPU utilisation on the broker as a percentage of one network or I/O thread.

Quotas (both types) are defined on a per-broker basis. In other words, a quota is enforced locally within an individual broker — irrespective of what the client may be doing on other brokers. If a client connects to two brokers, it will be served the equivalent of two quotas; a multiple of *N* quotas for *N* brokers. Even though a client may receive a multiple of its original quota, it can never exceed the original quota limit on any given broker.

The addition of quotas to Kafka was one of the earliest improvement proposals, implemented within KIP-13[43] and released in version 0.9.0.0. The decision to enforce quotas at the broker level rather than at the cluster level was done out of pragmatism — it was deemed too complicated to pool resource consumption metrics in real-time from all brokers to determine the aggregate consumption and to enforce the combined usage uniformly across all brokers. The consensus mechanism for tracking cross-broker resource usage was considered to be more difficult to implement than the quota enforcement mechanism. As such, the simpler solution was chosen instead, and despite numerous iterative enhancements, remains per-broker to this day.

The application of per-broker quotas suffers from one notable drawback: a client's effective (multiplied) quota dilates with the cluster size. The addition of nodes to a cluster (to meet increased demand or to increase durability) carries with it an increased likelihood that a client will connect to more brokers as a result, potentially leading to a higher quota multiple. At the same time, it cannot

[43] https://cwiki.apache.org/confluence/x/cpcWAw

be stated definitively that a client will connect to more brokers as the latter are scaled out — the actual number of broker connections will depend on the number of partitions for the topics that the client publishes to or consumes from, as well as the number of backing replicas. When scaling the cluster, it may be necessary to adjust each client's quota individually, projecting aggregate numbers and working back to determine what the per-broker quotas should be — compensating for the effects of the dilation.

Network bandwidth quotas

Network bandwidth quotas rely on the amount of transferred data as a definitive metric for assessing a client's utilisation of a broker's available resources. It may be thought of as a compound metric — increasing the rate of data transfer places a greater strain on the network, but also commensurately utilises the I/O channels on the broker, and may lead to increased memory pressure (due to buffering). In addition, when the client connects to an SSL listener, the broker loses its ability to employ the *zero-copy* optimisation, involving the CPU to encrypt and decrypt network data. With SSL enabled, the greater the transfer rate, the greater the load on the CPU.

Zero-copy describes computer operations in which the CPU does not perform the task of copying data from one memory area to another. In a typical I/O scenario, the transfer of data from a network socket to a storage device occurs without the involvement of the CPU and with a reduced number of context switches between the kernel and user mode.

A network bandwidth quota is specified as a pair of values: a `producer_byte_rate` and a `consumer_byte_rate` — representing the upper bound on the allowable bandwidth, in bytes per second (B/s). This unit of measurement is a slight departure to the conventional way of measuring bandwidth — bits per second; however, when dealing with application-level payload sizes, operating with byte multiples is more convenient.

Quotas are enforced by sampling the client's activity over a period of time, using a rudimentary *sliding window* algorithm. A pair of broker properties — `quota.window.num` and `quota.window.size.seconds` — stipulate the number of samples N retained and the duration of each sampling S period, respectively. The default values are `1` — for the sample duration, and `11` — for the number of samples. Collectively, the samples represent a sliding window.

The enforcement algorithm compares the client's observed resource utilisation U with the maximum allowed by the quota Q, over the observation period T. The precise calculation of T is somewhat intricate, taking into account the number of available samples, the age of the oldest sample and the amount of time spent in the current sampling period. For simplicity and convenience, assume that T is the result of multiplying of N by S. While this is not entirely accurate, the difference is at most S seconds; the simplified formula will suffice for the purpose of an explanation. We will return to the precise calculation later.

When U exceeds Q, the broker will penalise the client by introducing an artificial delay into the response. The client is not aware that a penalty is in force, observing what it believes is network

congestion. There are no errors as such, other than potential errors caused by network timeouts if the delay is in excess of the client's timeout tolerance. (This is possible, as we will shortly see.)

The calculation of the delay D is given by the following simple formula:

$$D = \frac{T(U - Q)}{Q}$$

Where the value of U is obtained by summing all values over the last `quota.window.num` samples, and T is the product of `quota.window.num` and `quota.window.size.seconds`.

The duty cycle Y of the system is given by:

$$Y = \frac{T}{D + T}$$

We can work through the formula using several examples. Assuming the write quota is set to 20 kB/s, with three clients *C0*, *C1* and *C2*, consuming at rates of 14 kB/s, 36 kB/s and 100 kB/s, respectively. The value for T will be *10* for simplicity.

For *C0*, $DC_0 = 10 \times (14 - 20) / 20 = -1$.

Since DC_0 is a negative number, and the delay must clearly be greater than zero, no penalty is applied in this case.

For *C1*, $DC_1 = 10 \times (36 - 20) / 20 = 8$.

That's eight seconds of delay before the response is returned. After the delay elapses, *C1* is free to publish the next batch. Assuming a constant publishing rate, *C1* will cycle between ten seconds of productivity, followed by eight seconds of hiatus — a duty cycle of just under 56%.

For *C2*, $DC_2 = 10 \times (100 - 20) / 20 = 40$.

A delay of forty seconds in this case is quite hefty, reducing the duty cycle to 20%. Still, the penalty is fitting, given the rate at which *C2* is attempting to publish.

In the case of a producer, the broker will append the request batch to the log but will not return a response immediately. Instead, the response is queued internally for the duration of the penalty time before being returned to the client. In the case of a consumer, the request is delayed for the duration of the penalty before performing the read.

In both cases, Kafka arranges the I/O operation relative to the delay such that the penalty does not cause undue memory pressure on the broker. When writing, the batch is appended to the log before pausing. When reading, the pause occurs before the disk fetch.

Request rate quotas

When bandwidth quotas were originally introduced, it was assumed that bandwidth is a holistic indicator of a client's impact on a broker's resource utilisation. And in many ways bandwidth is

still an excellent metric. The main issue with bandwidth metrics is that they focus on the payload, overlooking the request itself. And indeed, when the requests are reasonably sized and the clients are correctly configured (and well-behaved), the act of making the request carries a negligible overhead on the broker compared to servicing the request. However, if a client sends requests too quickly (for example, a consumer with `fetch.max.wait.ms` or `max.partition.fetch.bytes` set to very small values), it can still overwhelm the broker even though individual request/response sizes may be small.

Request rate quotas were introduced in KIP-124[44] as part of release 0.11.0.0, when it became obvious that an additional level of control was required to prevent denial of service from frequent protocol activity. In addition to protecting brokers from excessive request rates, the improvement proposal was generalised to address other scenarios that saw clients utilise the broker's CPU disproportionately to the request/response size. These include —

- Denial of service from clients that overload brokers with continuous unauthorized requests.
- Compression settings on the broker that contradict producers' assigned compression scheme, requiring decompression and re-compression on the broker.
- A mixture of TLS and non-TLS clients (where both listener types have been exposed); the former exert a significantly greater load on the broker for the same amount of traffic.

Request quotas are configured as a fraction of overall time a client is allowed to occupy request handler (I/O) threads and network threads within each quota window. For example, a quota of 50% implies that half of a thread can be utilised on average within the measured time window, 100% implies that a full thread may be utilised, 200% is the equivalent of two full-time threads, and so on.

The underlying mechanism for enforcing request rate quotas is virtually identical to the one used for network bandwidth quotas, with the only notable difference being the metric that is collected over the sampling window — using CPU time rather than the number of bytes sent or received.

The request rate calculations piggyback on the `quota.window.num` and `quota.window.size.seconds` properties, originally devised for network bandwidth quotas. The enforcement mechanism applies an artificial delay, using the formula presented earlier. The parity between the two mechanisms simplifies their management and reduces the number of tuning parameters.

Request rate quotas are effective at avoiding unintended DoS from misconfigured applications or defective client libraries, where a bug may inadvertently result in a high request rate. While this caters to some DoS scenarios, it does not completely insulate the cluster from all types of DoS attacks originating from malicious clients. A malicious actor may launch a distributed denial of service (DDoS) attack by recruiting a large number of connections, possibly originating from multiple hosts. This would result in a large amount of expensive authentication or CPU-intensive requests that may overwhelm the broker.

Ordinarily, the number of I/O and network threads is set in some proportion to the capabilities of the underlying hardware — typically, the number of CPU cores. One of the advantages of expressing

[44] https://cwiki.apache.org/confluence/x/y4IYB

request rates in absolute percentage terms (i.e. a fraction of one core) is that it pins the quota values, preventing the quotas from inadvertently dilating with the scaling of a broker's resources. For example, if the broker instances were to undergo a hardware refresh to quadruple the number of CPU cores, the existing absolute quotas would not be affected. Had the quotas been expressed in relative terms (i.e. a fraction of all cores), all existing request rate quotas would have been implicitly dilated by a factor of four. Note, the dilation-resistance of request rate quotas only applies to scaling in the vertical plane; in other words, when a broker's processing capacity is increased. When scaling horizontally — through the addition of broker nodes — the request rate quotas suffer from the same dilation effect that we saw with bandwidth quotas.

The main drawback of stating absolute values is that it requires the operator to be aware of the broker's available capacity — the number of I/O and network threads. These values are configured through the `num.io.threads` and `num.network.threads` broker properties and, as we saw in Chapter 9: Broker Configuration, can also be changed remotely. Due attention should be paid to both the contents of `server.properties` and the dynamic configuration; otherwise, the quotas may end up misconfigured — dispensing a different amount of resources to what was intended.

Subject affinity and precedence order

Quotas apply to two entity types: *user principals* and *client IDs*. The reader should by now be intimately familiar with users, which were covered in detail in Chapter 16: Security. Users are the subjects of authentication and authorization controls. Client IDs were briefly mentioned in Chapter 10: Client Configuration. A client ID is an optional, free-form logical identifier of a client connection, configured via the `client.id` client-side property. Client IDs allow Kafka to distinguish between variations of a client and are orthogonal to usernames.

Unlike usernames, whose use is policed by Kafka's authentication machinery, arbitrary client IDs can be presented at the discretion of the client. Client IDs may be used to further subcategorise clients that operate under a single username. Perhaps an application comprises multiple discrete processes that share the same Kafka credentials. These processes may be designated different tasks, which they carry out on behalf of the encompassing application. When the multiple different but related processes connect to Kafka, it is prudent to distinguish among them — maintaining fine-grained accounting and traceability of client actions.

Quotas are defined for either usernames or client IDs individually, or may cover a combination of the two. There are a total of eight ways a quota may be associated with its subject, depicted in the table below. Additionally, the table captures the precedence order of each association, with the lowest number corresponding to the highest priority.

Precedence order	Username partial association	Client ID partial association
1	Specific username	Specific client ID
2	Specific username	Default client ID
3	Specific username	Unspecified
4	Default username	Specific client ID
5	Default username	Default client ID
6	Default username	Unspecified
7	Unspecified	Specific client ID
8	Unspecified	Default client ID

Upon the consumption of a resource, Kafka will iterate through the table above in the order of increasing precedence number, stopping at the first rule that matches the client's attributes (being the username and client ID). If Kafka is unable to match any of the rules listed above, it will consult the `quota.producer.default` and `quota.consumer.default` broker properties to determine whether a static default has been set. These properties are deprecated in favour of dynamic configuration; it is recommended to avoid defining quota defaults in `server.properties` — instead, quotas should be managed exclusively using the `kafka-configs.sh` CLI or the equivalent Admin Client APIs. Finally, in the absence of a rule matching the supplied username and client ID, Kafka will not apply a quota to a connected client.

With the exception of the `quota.producer.default` and `quota.consumer.default` deprecated properties, quotas are persisted in ZooKeeper, under the `/config/users` and `/config/clients` znodes. Specifically, for the eight options above, the corresponding persistence schemes are:

1. Username + client ID combination: `/config/users/<username>/clients/<clientId>`
2. Username + default client ID: `/config/users/<user>/clients/<default>`
3. Username: `/config/users/<username>`
4. Default username + specific client ID: `/config/users/<default>/clients/<clientId>`
5. Default username + default client ID: `/config/users/<default>/clients/<default>`
6. Default username: `/config/users/<default>`
7. Client ID: `/config/clients/<clientId>`
8. Default client ID: `/config/clients/<default>`

With the ZooKeeper aspect of quota persistence being fully insulated from the operator, one might wonder where the knowledge of the storage hierarchy may be required. The answer speaks to troubleshooting. Should you encounter an irreconcilable difference between the behaviour of the cluster and the reported configuration, it may be necessary to jump into a ZooKeeper shell to diagnose the potential causes of the unexplained behaviour.

At this point, the reader would be right in pointing out that rules #2 and #3 look suspiciously similar, as do #4 and #7. There is a subtle difference among them, relating to how a quota is distributed among the connected clients. When the rule scopes both a username and a client ID, even if one or

both of those values are set to `<default>`, the quota will be allocated for the sole use of the connected client, as well as all other clients with identical username and client ID attributes.

Conversely, if a quota specifies either the username or the client ID, but not both, then the quota will be shared among all clients that match that username or client ID, whichever one was supplied, and irrespective of the value of the omitted attribute.

Understandably, this might cause a certain amount of confusion, which can be resolved with a few examples. We shall focus on rules #1, #2 and #3 because they capture scenarios where both attributes are assigned to specific values, where one is assigned and the other references the default entity, and finally, where one is assigned and the other is omitted. Consider the following quotas:

Username	Client ID	Read rate (kB/s)	Rule type
`alice`	`pump`	400	1
`alice`	`<default>`	300	2
`alice`	∅	200	3

Note, the value ∅ means that the attribute is not set; the value `<default>` implies the default entity for the attribute in question.

When a client *C0* with the username/client ID tuple (`alice`,`pump`) connects, it will be allocated 400 kB/s of read capacity — having matched the most specific rule.

When *C1* — a client with the same attributes (`alice`,`pump`) — subsequently connects, it will share the quota with *C0* — having matched the same rule. In other words, the two clients will not be able to consume more than 400 kB/s worth of data between them. If *C0* over-consumes, it will affect *C1* and *vice versa*.

A third client *C2* connects with (`alice`,`sink`) — matching the (`alice`,`<default>`) rule and qualifying for 300 kB/s of bandwidth.

A fourth client *C3* connects with (`alice`,`drain`) — matching the (`alice`,`<default>`) rule for 300 kB/s. However, although *C2* and *C3* were matched by the same rule, they have different client IDs and will not share the quota. In other words, *C2* and *C3* have 300 kB/s *each* of allowable bandwidth.

A fifth client *C4* connects with the same (`alice`,`drain`) attributes as *C3*. This client will end up sharing the bandwidth with *C3* — 300 kB/s among them.

Next, we have *C5* — a client authenticated as `alice`, having omitted the optional client ID. Despite what one might assume, this is matched by the (`alice`,`<default>`) rule. When a default entity is specified in a rule, it will match a client where no value for the corresponding attribute is provided. Essentially, at no point is the (`alice`,∅) rule fired; deleting it would make no difference.

C6 — another `alice` without a client ID, will share the quota with *C5*.

Finally, *C7* — a user authenticated as `bob` — will not match any of the rules in the table and will be allowed to operate with unbounded capacity.

Summarising the above examples, we have the table below:

Client	Username	Client ID	Matching rule	Bandwidth (kB/s)	Shared among
C0	alice	pump	(alice,pump)	400	C0, C1
C1	alice	pump	(alice,pump)	400	C0, C1
C2	alice	sink	(alice,<default>)	300	C2
C3	alice	drain	(alice,<default>)	300	C3, C4
C4	alice	drain	(alice,<default>)	300	C3, C4
C5	alice		(alice,<default>)	300	C5, C6
C6	alice		(alice,<default>)	300	C5, C6
C7	bob		Unmatched	Unbounded	C7

Imagine for a moment that we deleted the rule (alice,<default>), leaving just the (alice,pump) and (alice,□) rules. How would this affect the quota distribution?

Client	Username	Client ID	Matching rule	Bandwidth (kB/s)	Shared among
C0	alice	pump	(alice,pump)	400	C0, C1
C1	alice	pump	(alice,pump)	400	C0, C1
C2	alice	sink	(alice,□)	200	C2, C3, C4, C5, C6
C3	alice	drain	(alice,□)	200	C2, C3, C4, C5, C6
C4	alice	drain	(alice,□)	200	C2, C3, C4, C5, C6
C5	alice		(alice,□)	200	C2, C3, C4, C5, C6
C6	alice		(alice,□)	200	C2, C3, C4, C5, C6
C7	bob		Unmatched	Unbounded	C7

The first two clients — *C0* and *C1* — would be unaffected by the change, as they are matched by the most specific and, therefore, the highest priority rule. Clients *C2* through to *C6* would now be matched by the (alice,□) rule. Although they have different client IDs (clients *C2*, *C3* and *C4*) and in some cases, the client IDs have been omitted (*C5* and *C6*), all five clients will share the same bandwidth quota. Finally, *C7* is unaffected by the change.

Comparing the two sets of scenarios above, the difference between the (alice,<default>) and (alice,□) rules is not in the way they match clients. Both rules are equally eager in matching all clients authenticated as alice, irrespective of the value of the client ID, or whether client ID is provided. The difference between the two is in how they distribute the quotas. In the (alice,<default>) scenario, there are multiple quotas distributed evenly among matching clients having the same username and client ID attributes. In the (alice,□) scenario, there is a single quota shared among all clients with the matching username.

Precedence order operates irrespective of bandwidth or resource quotas. In the examples above, the most specific rules had the most relaxed bandwidth quotas. We could have just as easily applied the most conservative quota to the most specific rule — Kafka doesn't care either way. There is not a lot in the way of best-practices either. When quotas are employed as a security mechanism, it is best to start with the most conservative quota for the least specific rule — (`<default>`,`<default>`), increasing the quota with rule specificity. This approach is the equivalent of a *default-deny* security policy, where permission is granted, rather than taken away.

Applying quotas

Having ascertained how quotas are structured, persisted and applied, it is only fitting to set up an example that demonstrates their use. The only snag is that our previous examples, which published in small quantities, are hardly sufficient to trip a quota limit — unless the latter is absurdly conservative. Before we can demonstrate anything, we need a rig that is capable of generating a sufficient volume of records to make the experiment worthwhile.

The complete source code listing for the upcoming example is available at github.com/ekoutanov/effectivekafka[45] in the `src/main/java/effectivekafka/quota` directory.

The listing below is that of a producer that has been rigged to publish records at an unbounded speed.

```
import org.apache.kafka.clients.producer.*;

public final class QuotaProducerSample {
  public static void main(String[] args)
      throws InterruptedException {
    final var topic = "volume-test";

    final var config = new ScramProducerConfig()
        .withBootstrapServers("localhost:9094")
        .withUsername("alice")
        .withPassword("alice-secret")
        .withClientId("pump")
        .withCustomEntry(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG,
                         600_000);
```

[45] https://github.com/ekoutanov/effectivekafka/tree/master/src/main/java/effectivekafka/quota

```
    final var props = config.mapify();
    try (var producer = new KafkaProducer<String, String>(props)) {
      final var statsPrinter = new StatsPrinter();

      final var key = "some_key";
      final var value = "some_value".repeat(1000);

      while (true) {
        final Callback callback = (metadata, exception) -> {
          statsPrinter.accumulateRecord();
          if (exception != null) exception.printStackTrace();
        };

        producer.send(new ProducerRecord<>(topic, key, value),
                      callback);
        statsPrinter.maybePrintStats();
      }
    }
  }
}
```

This client does not use the traditional `Map<String, Object>` for building the configuration; instead, we have opted for the type-safe configuration pattern that was first introduced in Chapter 11: Robust Configuration. The complete listing of `ScramProducerConfig` is available in the GitHub repository. It has been omitted here as it bears little bearing on the outcome.

The value of the record is sized to be exactly 10 kilobytes. The record, combined with the key, headers, and other attributes will be just over 10 kB in size. The loop simply calls `Producer.send()` continuously, attempting to publish as much data as possible. The `StatsPrinter` class, shown below, helps keep track of the publishing statistics.

```
import static java.lang.System.*;

final class StatsPrinter {
  private static final long PRINT_INTERVAL_MS = 1_000;

  private final long startTime = System.currentTimeMillis();
  private long timestampOfLastPrint = startTime;
  private long lastRecordCount = 0;
  private long totalRecordCount = 0;

  void accumulateRecord() {
    totalRecordCount++;
```

```
  }

  void maybePrintStats() {
    final var now = System.currentTimeMillis();
    final var lastPrintAgo = now - timestampOfLastPrint;
    if (lastPrintAgo > PRINT_INTERVAL_MS) {
      final var elapsedTime = now - startTime;
      final var periodRecords = totalRecordCount - lastRecordCount;
      final var currentRate = rate(periodRecords, lastPrintAgo);
      final var averageRate = rate(totalRecordCount, elapsedTime);
      out.printf("Elapsed: %,d s; " +
                 "Rate: current %,.0f rec/s, average %,.0f rec/s%n",
                 elapsedTime / 1000, currentRate, averageRate);
      lastRecordCount = totalRecordCount;
      timestampOfLastPrint = now;
    }
  }

  private double rate(long quantity, long timeMs) {
    return quantity / (double) timeMs * 1000d;
  }
}
```

We won't dwell on the `StatsPrinter` implementation details; suffice it to say that it's a simple helper class that tracks the total number of published records and may be called periodically to output the rate over the last sampling period, as well as the blended average rate. This will be useful for comparing the production rate with and without quotas.

Before we can run this publisher, we need to create the `volume-test` topic and assign the `Write` operation to user `alice`. Run the pair of commands below.

```
$KAFKA_HOME/bin/kafka-topics.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --create --topic volume-test \
    --partitions 1 --replication-factor 1
```

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:alice \
    --operation Write --topic volume-test
```

Run the `QuotaProducerSample` sample, leaving it on for a couple of minutes.

```
Elapsed: 1 s; Rate: current 15 rec/s, average 15 rec/s
Elapsed: 2 s; Rate: current 1,162 rec/s, average 8 rec/s
Elapsed: 3 s; Rate: current 3,770 rec/s, average 393 rec/s
Elapsed: 4 s; Rate: current 4,948 rec/s, average 1,238 rec/s
Elapsed: 5 s; Rate: current 5,026 rec/s, average 1,981 rec/s
Elapsed: 6 s; Rate: current 4,851 rec/s, average 2,489 rec/s
Elapsed: 7 s; Rate: current 4,751 rec/s, average 2,827 rec/s
Elapsed: 8 s; Rate: current 7,537 rec/s, average 3,068 rec/s
...
(omitted for brevity)
...
Elapsed: 112 s; Rate: current 9,538 rec/s, average 8,593 rec/s
Elapsed: 113 s; Rate: current 9,145 rec/s, average 8,601 rec/s
Elapsed: 114 s; Rate: current 9,245 rec/s, average 8,606 rec/s
Elapsed: 115 s; Rate: current 9,547 rec/s, average 8,611 rec/s
Elapsed: 116 s; Rate: current 9,531 rec/s, average 8,620 rec/s
Elapsed: 117 s; Rate: current 9,384 rec/s, average 8,627 rec/s
Elapsed: 118 s; Rate: current 9,451 rec/s, average 8,634 rec/s
Elapsed: 119 s; Rate: current 8,779 rec/s, average 8,641 rec/s
Elapsed: 120 s; Rate: current 9,471 rec/s, average 8,642 rec/s
```

The publisher takes some time before settling into its maximum speed. In the example above, the unconstrained publisher was operating at around 9,000 records per second, each record being just over 10 kB big.

For our second run, we will apply a consumer rate quota on the combination of the user `alice` and the `pump` client ID. Quotas are configured using the `kafka-configs.sh` CLI, targetting either (or both) `users` and `clients` entity types.

```
$KAFKA_HOME/bin/kafka-configs.sh \
    --zookeeper localhost:2181 --alter --add-config \
    'producer_byte_rate=100000' \
    --entity-type users --entity-name alice \
    --entity-type clients --entity-name pump
```

```
Completed Updating config for entity: user-principal □
    'alice', client-id 'pump'.
```

Run the `QuotaProducerSample` again. With the per-second output, the effect of throttling is apparent. Kafka will allow for the occasional bursting of traffic, followed by a prolonged period throttling, before relaxing again. While the throughput might not be smooth, changing abruptly as a result of the intermittent penalties, the overall throughput quickly stabilises at 10 records per second (just over 100 kB/s) and stays that way for the remainder of the run.

```
Elapsed: 1 s; Rate: current 7 rec/s, average 7 rec/s
Elapsed: 2 s; Rate: current 102 rec/s, average 3 rec/s
Elapsed: 3 s; Rate: current 1 rec/s, average 35 rec/s
Elapsed: 4 s; Rate: current 1 rec/s, average 26 rec/s
Elapsed: 5 s; Rate: current 1 rec/s, average 21 rec/s
Elapsed: 6 s; Rate: current 1 rec/s, average 18 rec/s
Elapsed: 7 s; Rate: current 1 rec/s, average 15 rec/s
Elapsed: 8 s; Rate: current 1 rec/s, average 13 rec/s
Elapsed: 9 s; Rate: current 1 rec/s, average 12 rec/s
Elapsed: 10 s; Rate: current 1 rec/s, average 11 rec/s
Elapsed: 11 s; Rate: current 1 rec/s, average 10 rec/s
Elapsed: 13 s; Rate: current 1 rec/s, average 9 rec/s
Elapsed: 14 s; Rate: current 96 rec/s, average 9 rec/s
Elapsed: 15 s; Rate: current 2 rec/s, average 14 rec/s
Elapsed: 16 s; Rate: current 1 rec/s, average 13 rec/s
Elapsed: 17 s; Rate: current 1 rec/s, average 13 rec/s
Elapsed: 18 s; Rate: current 1 rec/s, average 12 rec/s
Elapsed: 19 s; Rate: current 1 rec/s, average 11 rec/s
Elapsed: 20 s; Rate: current 1 rec/s, average 11 rec/s
Elapsed: 21 s; Rate: current 1 rec/s, average 10 rec/s
Elapsed: 23 s; Rate: current 1 rec/s, average 10 rec/s
Elapsed: 24 s; Rate: current 1 rec/s, average 9 rec/s
Elapsed: 25 s; Rate: current 96 rec/s, average 9 rec/s
Elapsed: 26 s; Rate: current 2 rec/s, average 12 rec/s
Elapsed: 27 s; Rate: current 1 rec/s, average 12 rec/s
Elapsed: 28 s; Rate: current 1 rec/s, average 12 rec/s
Elapsed: 29 s; Rate: current 1 rec/s, average 11 rec/s
Elapsed: 30 s; Rate: current 1 rec/s, average 11 rec/s
Elapsed: 32 s; Rate: current 1 rec/s, average 10 rec/s
Elapsed: 33 s; Rate: current 1 rec/s, average 10 rec/s
Elapsed: 34 s; Rate: current 1 rec/s, average 10 rec/s
Elapsed: 35 s; Rate: current 1 rec/s, average 10 rec/s
Elapsed: 36 s; Rate: current 95 rec/s, average 9 rec/s
...
```

```
(omitted for brevity)
...
Elapsed: 114 s; Rate: current 93 rec/s, average 10 rec/s
Elapsed: 115 s; Rate: current 2 rec/s, average 10 rec/s
Elapsed: 116 s; Rate: current 1 rec/s, average 10 rec/s
Elapsed: 117 s; Rate: current 1 rec/s, average 10 rec/s
Elapsed: 118 s; Rate: current 1 rec/s, average 10 rec/s
Elapsed: 120 s; Rate: current 1 rec/s, average 10 rec/s
```

In addition to the `producer_byte_rate` quota used in the previous example, it is also possible to specify a `consumer_byte_rate` and a `request_percentage`. For example, the following sets all three in the one command:

```
producer_limit="producer_byte_rate=1024" && \
consumer_limit="consumer_byte_rate=2048" && \
request_limit="request_percentage=200" && \
limits="${producer_limit},${consumer_limit},${request_limit}" && \
$KAFKA_HOME/bin/kafka-configs.sh \
    --zookeeper localhost:2181 --alter --add-config $limits \
    --entity-type users --entity-name alice
```

Similarly, it is possible to remove multiple limits with one command:

```
$KAFKA_HOME/bin/kafka-configs.sh \
    --zookeeper localhost:2181 --alter --delete-config \
    "producer_byte_rate,consumer_byte_rate,request_percentage" \
    --entity-type users --entity-name alice
```

To list the limits, run `kafka-configs.sh` with the `--describe` flag:

```
$KAFKA_HOME/bin/kafka-configs.sh \
    --zookeeper localhost:2181 --describe \
    --entity-type users
```

This is where `kafka-configs.sh` might get a little unintuitive. Listing the configuration with `--entity-type users` brings up those configuration entries that only feature a user, excluding any entries that contain both the user and the client. Therefore, our existing quota set for a combination of user `alice` and client ID `pump` will not be listed. To list quotas that apply to the combination of users and clients, pass both entity types to the command:

```
$KAFKA_HOME/bin/kafka-configs.sh \
    --zookeeper localhost:2181 --describe \
    --entity-type users --entity-type clients


Configs for user-principal 'alice', client-id 'pump' ␣
    are producer_byte_rate=100000
```

Buffering and timeouts

An observant reader would have picked up a minor detail in the configuration of the producer client, which did not appear in any prior examples in this book. Specifically, the following line:

```
.withCustomEntry(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG, 600_000);
```

This overrides the `delivery.timeout.ms` property from its default value of two minutes, setting it to ten minutes. For the sake of an experiment, remove the setting and run the example again. Just after two minutes into the run, the following error will be printed to the console.

```
org.apache.kafka.common.errors.TimeoutException: Expiring 1 ␣
    record(s) for volume-test-0:120007 ms has passed since ␣
    batch creation
```

To understand why this is happening, consider another client-side property `buffer.memory`, which defaults to `33554432` (32 MiB). This property sets an upper bound on the amount of memory used by the transmission buffer. If records are buffered faster than they can be delivered to the broker, the producer will eventually block when the buffer becomes full.

Given the size of each record is a smidgen over 10 kB, the number of records that can fit into the buffer is just under 3,355. In other words, a sufficiently fast producer can queue up to 3,355 records before the first record reaches the broker. Assuming a producer operating at full blast, the buffer will be filled almost immediately after starting the producer. (Given there are no `Thread.sleep()` calls in the `while` loop, the producer will be generating records at the maximum rate.) At an average throughput of 10 records/second, it will take the publisher 335 seconds to get through one complete allotment — which is just over five and a half minutes. With the default delivery timeout of two minutes, it is hardly a surprise that the error appears just after the 120-second mark. The delivery timeout was set to ten minutes to allow for sufficient slack over the minimum timeout, ensuring that the timeout does not occur under a normal operating scenario.

Kafka offers two obvious solutions to the problem of timeouts, where the buffer churn takes longer than the delivery timeout. We have just discussed the first — *increase the timeout*. The second is its flip side — *decrease the buffer size*.

The second approach is demonstrated in the `BufferedQuotaProducerSample`, listed below.

```
import static java.lang.System.*;

import org.apache.kafka.clients.producer.*;

public final class BufferedQuotaProducerSample {
  public static void main(String[] args)
      throws InterruptedException {
    final var topic = "volume-test";

    final var config = new ScramProducerConfig()
        .withBootstrapServers("localhost:9094")
        .withUsername("alice")
        .withPassword("alice-secret")
        .withClientId("pump")
        .withCustomEntry(ProducerConfig.BUFFER_MEMORY_CONFIG,
                         100_000);

    final var props = config.mapify();
    try (var producer = new KafkaProducer<String, String>(props)) {
      final var statsPrinter = new StatsPrinter();

      final var key = "some_key";
      final var value = "some_value".repeat(1000);

      while (true) {
        final Callback callback = (metadata, exception) -> {
          statsPrinter.accumulateRecord();
          if (exception != null) exception.printStackTrace();
        };

        final var record = new ProducerRecord<>(topic, key, value);
        final var tookMs = timed(() -> {
          producer.send(record, callback);
        });
        out.format("Blocked for %,d ms%n", tookMs);
        statsPrinter.maybePrintStats();
      }
    }
  }

  private static long timed(Runnable task) {
    final var start = System.currentTimeMillis();
    task.run();
```

```
    return System.currentTimeMillis() - start;
  }
}
```

The first change is the removal of the `ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG` property, and the use of the `ProducerConfig.BUFFER_MEMORY_CONFIG` property in its place, set to 100 kB. This enables the application to fit just under 10 records in the buffer, before the latter results in the blocking of `Producer.send()`.

The second change is mostly for illustrative purposes: The example was enhanced with timing code to show the effect of blocking in the `send()` method. Every call to `send()` is followed by a printout of the number of milliseconds that the method blocked for.

```
Blocked for 430 ms
Blocked for 1 ms
Blocked for 0 ms
Blocked for 0 ms
...
(omitted for brevity)
...
Blocked for 0 ms
Blocked for 28 ms
Blocked for 98 ms
Blocked for 102 ms
Elapsed: 1 s; Rate: current 97 rec/s, average 97 rec/s
Blocked for 1,103 ms
Elapsed: 2 s; Rate: current 4 rec/s, average 40 rec/s
Blocked for 1,099 ms
Elapsed: 3 s; Rate: current 1 rec/s, average 30 rec/s
Blocked for 1,103 ms
Elapsed: 4 s; Rate: current 1 rec/s, average 23 rec/s
Blocked for 1,100 ms
Elapsed: 5 s; Rate: current 1 rec/s, average 19 rec/s
Blocked for 1,097 ms
Elapsed: 7 s; Rate: current 1 rec/s, average 16 rec/s
Blocked for 1,104 ms
Elapsed: 8 s; Rate: current 1 rec/s, average 14 rec/s
Blocked for 1,097 ms
Elapsed: 9 s; Rate: current 1 rec/s, average 13 rec/s
Blocked for 1,105 ms
Elapsed: 10 s; Rate: current 1 rec/s, average 11 rec/s
Blocked for 1,100 ms
Elapsed: 11 s; Rate: current 1 rec/s, average 10 rec/s
```

```
Blocked for 1,099 ms
Elapsed: 12 s; Rate: current 1 rec/s, average 9 rec/s
Blocked for 1 ms
Blocked for 1 ms
Blocked for 1 ms
Blocked for 1 ms
...
(omitted for brevity)
...
Blocked for 0 ms
Blocked for 1 ms
Blocked for 1 ms
Blocked for 50 ms
Blocked for 1,097 ms
Elapsed: 13 s; Rate: current 82 rec/s, average 9 rec/s
Blocked for 1,103 ms
Elapsed: 14 s; Rate: current 1 rec/s, average 15 rec/s
Blocked for 1,106 ms
Elapsed: 16 s; Rate: current 1 rec/s, average 14 rec/s
Blocked for 1,097 ms
Elapsed: 17 s; Rate: current 1 rec/s, average 13 rec/s
Blocked for 1,097 ms
Elapsed: 18 s; Rate: current 1 rec/s, average 12 rec/s
Blocked for 1,104 ms
Elapsed: 19 s; Rate: current 1 rec/s, average 12 rec/s
Blocked for 1,097 ms
Elapsed: 20 s; Rate: current 1 rec/s, average 11 rec/s
Blocked for 1,103 ms
Elapsed: 21 s; Rate: current 1 rec/s, average 11 rec/s
Blocked for 1,103 ms
Elapsed: 22 s; Rate: current 1 rec/s, average 10 rec/s
Blocked for 1,097 ms
Elapsed: 23 s; Rate: current 1 rec/s, average 10 rec/s
Blocked for 2 ms
Blocked for 0 ms
Blocked for 0 ms
Blocked for 0 ms
...
(omitted for brevity)
...
```

There's an initial blocking of just over 400 ms, spent waiting on the cluster metadata. Once the metadata has been retrieved, subsequent `send()` methods are non-blocking, as the buffer is being

drained at the maximum speed that the broker is capable of. After around a hundred or so records, the quota limits come into effect and impose an artificial delay on the producer. At this point the buffer will fill up rapidly, causing subsequent blocking. Records are being acknowledged at a rate of approximately one every second, which reflects on the blocking time. After the penalty wears off, the producer bursts again for another 100 (approximately) records — again leading to a free-flowing `send()`. This cycle repeats roughly every 100 records.

Sensing quota enforcement

One of the well-known challenges of Kafka's quota implementation is the lack of explicit communication of the quota's enforcement from the broker to the offending (in a manner of speaking) client. Recall, a quota may be breached by a well-behaved client; the cause being none other than an attempt to produce or consume at a rate greater than what the broker is willing to support. The delay introduced by the broker is largely driven by a combination of two intentions: maintaining compatibility with older client versions and simplifying the client implementation, which remains agnostic to the goings-on in the broker.

The drawback of Kafka's implementation of quotas is that the client observes what it believes is degraded performance. In the more extreme cases, when the time penalty is substantial or the number of backlogged records is high, the client will begin to time out. This can result in the compounding of retry attempts, similar to the effects of a *congestive collapse*, discussed in Chapter 12: Batching and Compression. What Kafka lacks is a mechanism for signalling to the client that it is either approaching the limit of the quota or is in breach, so that the client can act at its discretion (independent of broker-side enforcement).

The conventional way of solving this problem, as we have seen earlier, is to limit the buffer size using the `buffer.memory` property — creating backpressure on the client. This works well in the majority of cases, particularly when dealing with non-time-sensitive data. On the other hand, if the client needs to know whether it is being throttled, there is no out-of-the-box mechanism to determine this, nor is there a way of probing the amount of available buffer capacity. Kafka offers no `Producer.trySend()` method that publishes a record conditional on the available channel capacity.

One scenario where blocking behaviour is undesirable is where a producer needs to publish heterogeneous data with mixed quality of service characteristics onto a single topic. The client might be collecting data from multiple sources, but there may be a clear need to emit certain types of events over others. For example, the client may need to periodically transmit critical status updates. At the same time, it may transmit other metrics which are useful to downstream consumers, but it is not critical that these metrics are delivered in a timely manner. (This scenario assumes that, for whatever reason, it is not feasible to separate the data into multiple streams with separate QoS guarantees; otherwise, this is addressed by the use of topic-level quotas.)

Where blocking behaviour is unacceptable, the client may choose to track the in-flight records and correlate these to the received acknowledgements, keeping a 'pending records' counter. The counter will increase as a result of throttling, up to the number of buffered records, and will revert to a

near-zero value when the limit is lifted. By observing this counter, the application can infer the approximate likelihood of a queued record being sent immediately. Alternatively, the counter may be replaced with the timestamp of the most recent unacknowledged record. (The timestamp is cleared when the last record has been acknowledged). By comparing the wall clock to this timestamp, the application is again able to infer the presence of throttling and act accordingly. Both methods are probabilistic; there is no categorical way to determine whether a throttle is in force, or whether the delayed acknowledgements are a result of genuine congestion. At any rate, both are indicative of degraded behaviour.

Tuning the duration and number of sampling windows

As it was previously stated, the quota enforcement algorithm represents a sliding window T comprising N samples wide (given by `quota.window.num`), each being S seconds long (given by `quota.window.size.seconds`). By default, these values are `11` and `1` respectively, implying that the sliding window spans at most eleven seconds of observed utilisation. In reality, the calculation of the window size subtracts the current time from the oldest in the set of retained samples. When the wall clock approaches the boundary of the current period, the window size T approaches the product of N and S. However, when the current sample rolls over to the next, the oldest sample used in the previous calculation is dropped, leaving the next oldest sample — S seconds later than its predecessor. This creates a sort of a *snapback effect*, where the T instantaneously goes from being $N \times S$ seconds wide, to $(N - 1) \times S$.

Looking at the output of the most recent example, there is a clear spike of throughput, followed by a period of suppressed (but not entirely quiesced) activity, which seems to recur in a predictable cycle. To understand the reason for this comb-like shape, consider what happens at various points in the client-broker interaction.

Given a window of 11 seconds at its peak width and a set rate limit of 100 kB/s, the client should, in theory, be allowed to produce up to 1,100 kB of data in any given eleven-second window. The window is important; although the quota is stated in bytes per second, the enforcement algorithm operates in terms of the current window size — varying between ten and eleven seconds. Following the initial metadata retrieval, the client is free to produce a volume of traffic that does not breach the threshold within the observed window. At full speed, this allowance will be exhausted in a matter of milliseconds. In fact, the client will overproduce by some margin before the limit kicks in, as the delay is a function of the difference between the observed utilisation and the allowed utilisation.

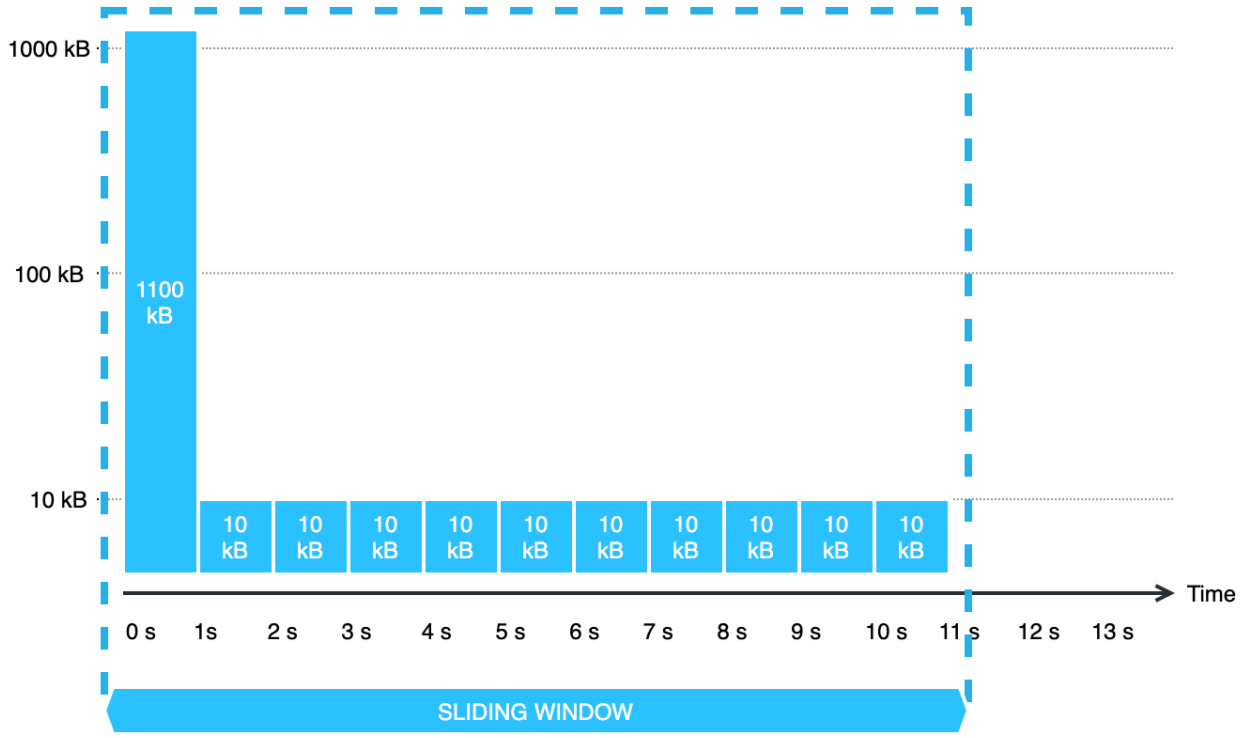
At this point, the client is being delayed by a small amount of time, as the extent of overproduction is quite minimal. Delayed does not mean stopped; the client will transmit the next batch after the broker's response is received. The batch size is given by the `batch.size` client property, which defaults to 16 KiB — just enough to fit one record. The client will be gently throttled until the conclusion of the first sampling period, where T approaches 11 seconds. By this time, the client would have produced just over 1,100 kB of data.

At the commencement of the second period, the value of T will abruptly snap back to 10 seconds.

Where the client had previously only barely violated the quota, the discontinuity in T now means that the quota has been breached by a greater extent. Specifically, the new delay D is given by:

$$D = 10 \times (1100 - 1000) / 1000$$

With the updated calculation, the delay is now one second. This is a 10% breach for a ten-second window, penalised by the duration of one sampling period. In other words, the next request will be delayed by one second before the broker responds. Incidentally, this amount of delay lines up with the commencement of the next sampling period, at which point the client will again be in breach of the quota by 10%, leading to another one-second penalty, and so on. The client will settle into a rhythm where it manages to send roughly 10 kB/s. This is illustrated in the diagram below, showing the amount of data transferred within each sampling period.

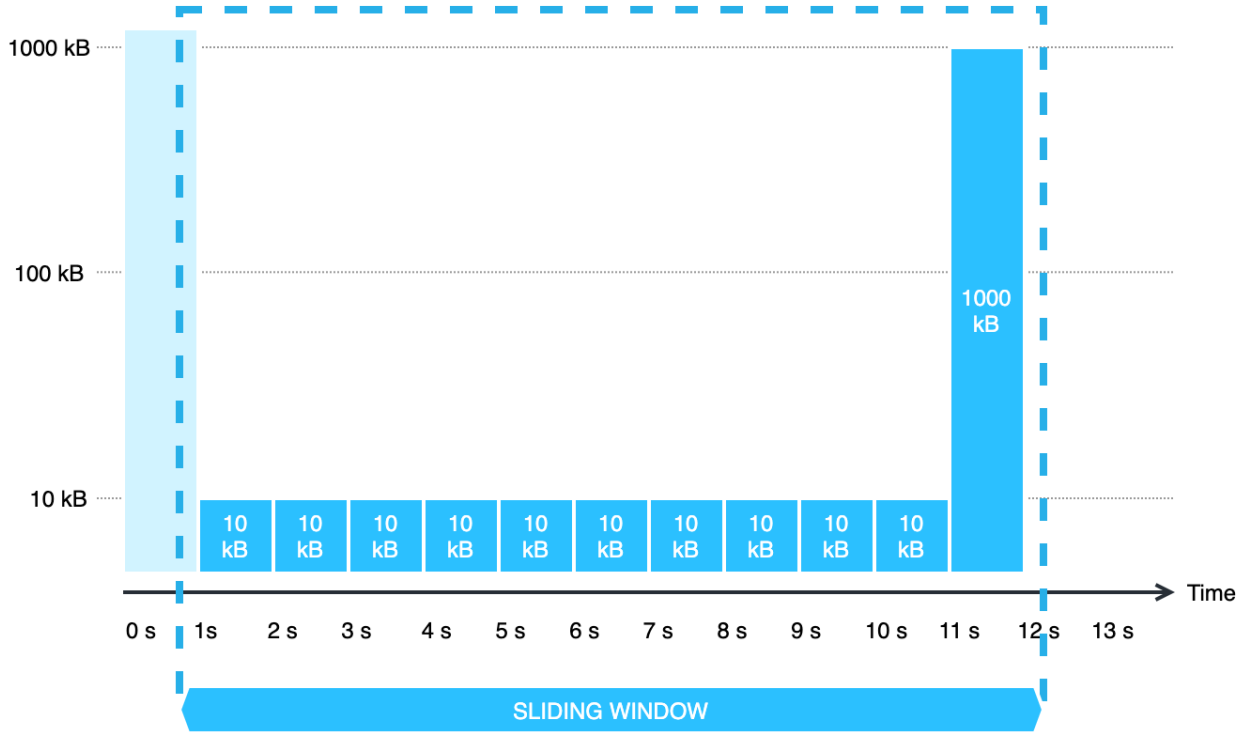


Sliding window — first cycle

The illustration is only a rough guide, being a simplified approximation; the area under the real bandwidth curve will not be so evenly rationed among the sampling periods. Still, it's a fair indication of what happens.

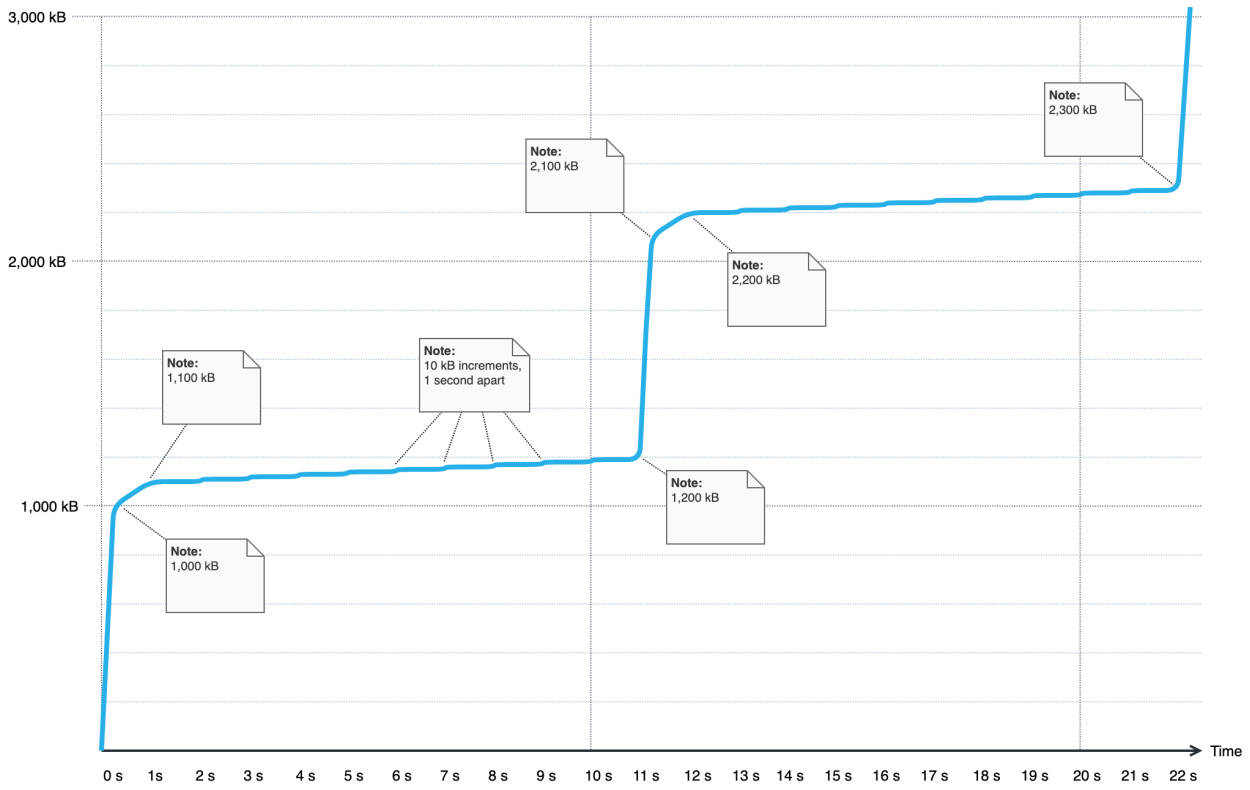
Upon the conclusion of the tenth second, we will have one sampling period with a high average utilisation value, followed by ten periods with relatively low utilisation values. When the 12th sampling period commences on the 11th second, the sliding window will have advanced to the point where the first sampling period is no longer in its scope. At this point, the aggregate utilisation across

the window will suddenly drop to around 100 kB — 900 kB short of the allotted maximum for a ten-second window. This explains the next traffic spike — the client takes the opportunity to saturate the broker with free-flowing traffic, lasting for just as long as the quota is breached once again, at which point the traffic is moderated with a series of small delays. At the commencement of the 13th sampling period, the window will snap back from 11 to 10 seconds, leaving the client in breach of the quota by 10%, slapping down a one-second penalty. It's easy to how this pattern repeats when the second peak drops off the scope of the sliding window on the 22nd second. Note, the second peak is approximately 10 kB shorter than the first peak, with all subsequent peaks being the same height. The effect of dropping the first sample from the sliding window, marking the commencement of the second cycle is depicted below.



Sliding window — second cycle

The final diagram depicts the aggregate utilisation over a series of sampling periods, covering two complete cycles. The diagram calls out the critical inflexion points that were discussed in the narrative above.



Aggregate utilisation

Recalling the earlier-stated formula relating the delay to the duty cycle, one might assume that the cyclic shape of the bandwidth and the duty cycle are somehow equivalent or at least comparable. Curiously, this is not the case: The duty cycle relates to an individual penalty and represents the ratio of the request time to the duration of one request-response cycle. It can be thought of as operating at the *micro* level. By comparison, the *macro-level* cycle that is trivially observable — the one we have just been discussing — is a function of the length of the sliding window and the duration of each sampling period.

This brings us to the next point: how do the `quota.window.num` and `quota.window.size.seconds` properties (the values N and S) affect the behaviour of the system?

By increasing `quota.window.num`, the overall duration of the window T is increased, elevating the tolerance for bursty traffic. In the example of the default window (ranging from ten to eleven seconds), the system tolerated a burst of just over eleven times the prescribed limit. It did so at the expense of traffic symmetry. Conversely, reducing the overall window duration leads to a flatter traffic profile, having less tolerance for short periods of heightened activity. The reader may want to try changing the `quota.window.num` setting in `server.properties` — increasing and decreasing the duration of the window and rerunning the example on each occasion. The behaviour will be confirmed. And either way, the long-term average throughput will remain the same.

By increasing `quota.window.size.seconds`, the granularity of the sampling process is decreased, thereby inflating the magnitude of the snapback effect. When the snapback occurs, the difference

between the previous excess utilisation and the new excess utilisation is greater, leading to a proportionally longer penalty delay. For example, when the N is 11 and S is 3 seconds, the size of the window will vary between 30 seconds and 33 seconds at the two extremes of a sampling period. When a snapback occurs, the extent of the delay will be three seconds (the value of S).

There is one 'gotcha' in this section worth mentioning. The discussion above depicts the actual behaviour of the Kafka broker. The calculation of the window size and the snapback effect are not discussed in the official Kafka documentation, nor are they detailed in the corresponding KIPs. The detailed explanation of the shaping behaviour was produced as a result of analysing the source code of the broker implementation. This implementation is *unwarranted* and is subject to change without notice. For example, the maintainers of the project may fix the snapback issue by using a constant value for T or by interpolating the value of the oldest sampling period in proportion to the wall clock's position in the most recent sampling period. Either way, rectifying the snapback eliminates the corresponding discontinuity, resulting in a smoother traffic shape. (More frequent, shorter lasting penalties.) Of course, the snapback is just one aspect of the implementation that might be altered without conflicting with the documentation. As such, you should not build your application to rely on a specific characteristic of the traffic shape when a quota is enforced. The only aspect of the traffic shape that can be relied upon is the average bandwidth.

While the effect of `quota.window.num` is well understood and the setting may be reasonably altered in either direction, the effect of `quota.window.size.seconds` is more nuanced, subject to idiosyncratic behaviour that may be changed. There is no compelling reason why the sample period should be increased; if anything, keeping it low leads to less pronounced discontinuities in the resulting throughput. The default value of `1` is already at its permitted minimum.

In this chapter, we looked at one of Kafka's main mechanisms for facilitating multitenancy. Quotas support this capability by mitigating denial of service attacks, allowing for capacity planning and addressing quality of service concerns.

Quotas may be set in terms of allowable network bandwidth and request rates, applying either to a user principal or a client ID, or a combination of both. The enforcement algorithm works by comparing the client's resource utilisation to the set maximum, over a sliding window. Where the utilisation exceeds the set limit, the broker delays its responses, forcing the client to slow down.

Finally, we looked at the parameters behind the quota enforcement algorithm and explored their effect on the resulting traffic shape. The relationship is nuanced, where the resulting characteristics of the traffic profile are tightly coupled to the implementation and are difficult to infer. The more dependable attribute to reason about is that widening the sliding window allows for increased tolerance to bursty traffic.

In some ways, the quota mechanism is an extension of the security controls discussed in Chapter 16: Security; in other ways, it may be perceived as a standalone capability that bears merit in its own

right. Collectively, the combination of encryption, authentication, authorization and quotas ensure a safe and equitable operating environment for all users of a Kafka cluster.

Chapter 18: Transactions

When discussing event stream processing application, one topic of conversation that invariably comes up is that of delivery guarantees.

To recount our journey thus far, Chapter 3: Architecture and Core Concepts has introduced the concept of delivery semantics — distinguishing between *at-most-once* and *at-least-once* models and how either can easily be achieved in a Kafka consumer by changing the relative order of processing a record and confirming its offsets.

We subsequently looked at the notions of *idempotence* and *exactly-once* delivery in Chapter 6: Design Considerations — namely, the role of the client application in ensuring that the effects of processing a record are not repeated if the same record were to be consumed multiple times.

Finally, Chapter 10: Client Configuration demonstrated the use of the `enable.idempotence` producer property to ensure that records are not persisted in duplicate or out-of-order in the face of intermittent errors.

This chapter looks at one last control made available by Kafka — *transactions*. Transactions fill certain gaps of idempotent consumers with respect to the side-effects of record processing, and *enable end-to-end, exactly-once transactional semantics across a series of loosely-coupled stream processing stages*.

Preamble

Before we get too involved into unravelling the joys of transactional event processing, it is worth taking a moment to contemplate the extent of the work that has gone into its design and construction, and acknowledge the colossal efforts of numerous contributors who have made this possible. The KIP comprised around 60 individual work items and was planned for over three years. The final design was subject to a nine-month period of public scrutiny and evolved substantially as a result of community feedback. The design was one of the most profound changes to Kafka since its public release, delivering in excess of 15,000 lines of unit tests alone. Crucially, the performance overhead of transactions was kept to a minimum — subtracting an average of three to five per cent from the throughput of an equivalent non-transactional producer.

Transactions arrived in release 0.11.0.0, as part of a much larger KIP-98[46]. The significance of this may not be immediately apparent, but the reader has already witnessed this KIP in action in Chapter 10:

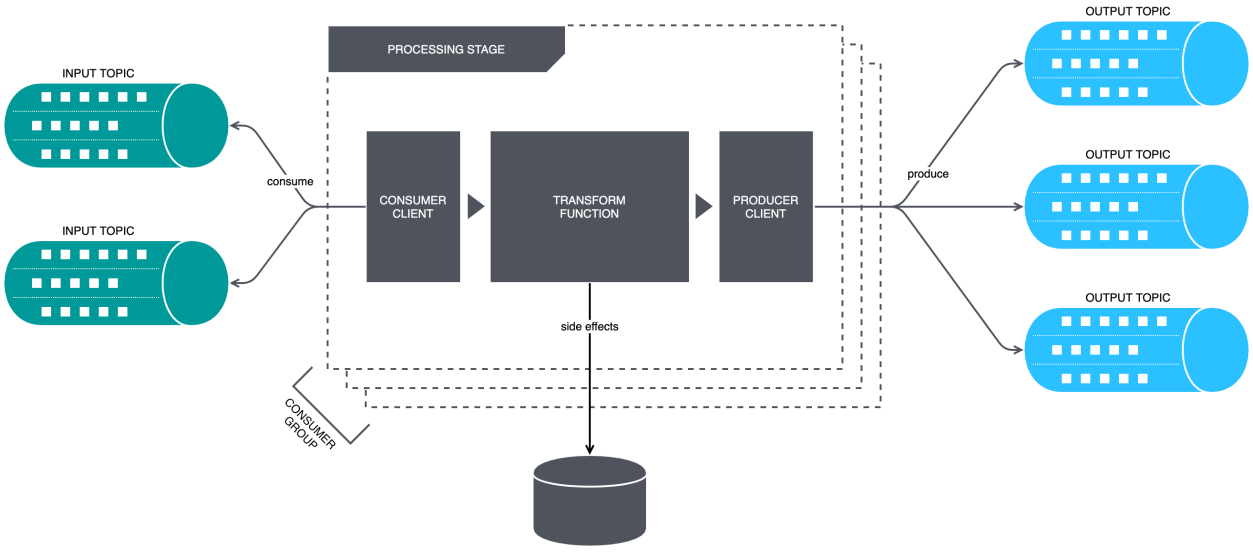
[46] https://cwiki.apache.org/confluence/x/ASD8Aw

Client Configuration, and most of our examples have, in fact, utilised the capabilities introduced by this KIP — namely, the `enable.idempotence` property. That's correct, both the *idempotent producer* and *transactional messaging* features are highly related and share a great deal in common. So in effect, the forthcoming study of transactional messaging might be seen as closing off the discussions started many chapters ago.

The rationale behind transactions

As it has been extensively discussed in the introductory chapters, the role of Kafka as an Event Streaming Platform is to facilitate the distribution and processing of events within a broader, Event-Driven Architecture.

Event streaming systems can be visualised as a set of loosely coupled actors that form a directed acyclic graph (DAG), where the nodes of the graph are processes that interact with Kafka topics, and the edges are the topics themselves. The processes, also called *stages*, may act either as a producer or a consumer, or a combination of both *vis à vis* one or more topics and client instances. A stage may also interact with other resources outside of Kafka, such as databases, APIs, and so forth. A reference model of a single stream processing stage and its neighbouring topics is depicted in the diagram below.



Stream processing stage

Recall the discussion on exactly-once delivery in Chapter 6: Design Considerations. To quote from the chapter:

> To achieve the coveted *exactly-once* semantics, consumers in event streaming applications must be *idempotent*. In other words, processing the same record repeatedly should have no net effect on the consumer ecosystem. If a record has no additive effects, the consumer is inherently idempotent. [...] Otherwise, the consumer must check whether a record

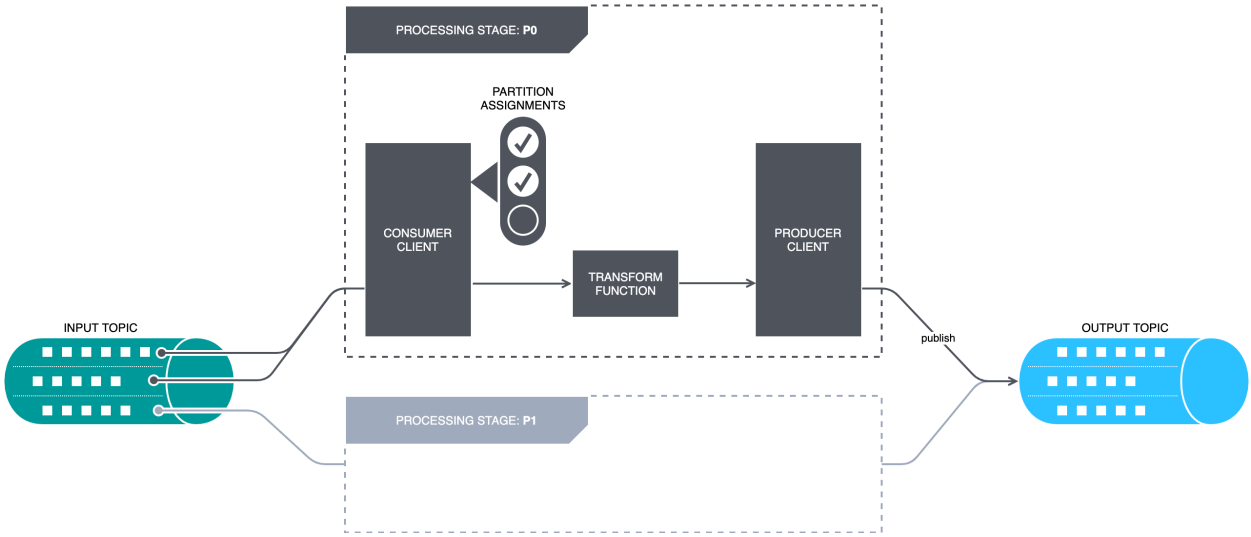
has already been processed, and to what extent, prior to processing the record. *The combination of at-least-once delivery and consumer idempotence collectively leads to exactly-once semantics.*

The problem — duplicate records

The ability to check and selectively carry out an action works well with databases, given that just about every database in existence supports the notion of a secondary index. (Put simply, a way of efficiently accessing records by means of some piece of information other than a primary key.) Kafka, although often likened to a database, does not support user-defined secondary indexes. The primary index of a record is the topic-partition-offset triple; in addition, Kafka allows us to index records by their timestamp for convenience. What Kafka lacks is the ability to nominate some attribute of a record that may be used to efficiently retrieve the record, or at least ascertain the record's presence.

Speaking of databases, Kafka was originally designed as a durable record store. Kafka's replication mechanism ensures that, with the appropriate configuration in place, once a record is acknowledged on the broker, that record is persisted in a manner that survives a broad range of failures. Aside from the *durability* guarantee, Kafka did not address any of the other attributes of *ACID* (Atomicity, Consistency, Isolation, Durability). We will touch on ACID again shortly.

Without a secondary index, how can a processing stage ensure that a record has not been published, for some input record that might be processed repeatedly as a result of at-least-once delivery? To illustrate the problem, consider a simple stage that transforms an input stream to an output stream using a straightforward mapping function:



Simple stream processing example

To further distil the problem, let's assume that a single 'input' record trivially maps to one 'output' record. Assume the input topic is a set of integers and the transform function simply squares each integer before forwarding it on. Contrived as it may be, this example serves to demonstrate the problem of attempting to apply idempotence to a storage medium that does not inherently support it. The topic topology and the transform function are not important; what is important is the fundamental property that there is exactly one output record corresponding to a processed input record, despite the prevalent at-least-once behaviour. And this property must be upheld in the presence of multiple instances of the transformation stage, operating within an encompassing consumer group.

Under a conventional *consume-transform-produce* model, the consumer side of a stage will read a record from the input topic, apply a transformation, and publish a corresponding record on the output topic, via a dedicated producer client. Once it receives an acknowledgement from the leader that the record has been durably persisted, it will commit the offsets of the input record. For simplicity, assume the producer is operating with idempotence enabled, being the recommended mode for most applications.

Consider the types of failures one might encounter with the model above. Putting aside buggy and misbehaving applications (and more complex Byzantine faults), at minimum, we would need to account for the failure of brokers, the abrupt termination of the transform process, and network connectivity issues. On the consumer side, network and broker failure can be dealt with by the retry mechanism embedded into `Consumer.poll()`. On the producer end, the use of idempotence will deal with broker and network failures, provided the failure does not last longer than the delivery timeout. This leaves us with the most taxing failure scenario: process failure. The outcome will depend on the exact point at which the process fails. Consider the options:

1. Failure before consumption.
2. Failure after consumption, but before the transform.
3. Failure following the transform, but before publishing.
4. Failure after publishing, but before confirming the offsets.
5. Failure after committing the offsets.

Case #1 is benign: nothing has yet happened and so the recovery is trivial. Case #2 is like #1, because Kafka (unlike its message queuing counterparts) does not update the state of the topic after a record has been read. Assuming the transform operation is a pure function that entails no state change of its own, case #3 is also a non-issue; the recovering process will replay the consumption and transform steps. Case #4 is where it gets interesting, being the first point where the failure occurs after a state change. The recovering process, having no awareness of the prior publish step will repeat all steps, resulting in two output records for the same input record. Finally, case #5 is again benign, as the committing of the offsets marks the completion of the process cycle.

Analysis of the above shows that, in the worst case, multiple output records may be written for a single input record; in the best case, one output record will be written. Assuming the input record is eventually processed, resulting in its offsets committed, at no point after the commit will there

be an observable absence of an output record. And that is the essence of at-least-once delivery. All prior material in this book has consistently guided the reader towards this direction.

We could have easily expanded upon the example above to include multiple input and output topics, and more complex transformation logic with possible side-effects. Fundamentally, this would not have changed the basic failure scenarios.

This basic formula for at-least-once delivery extends to any other stage that might be downstream of the one being considered. Provided an application-level identifier is included in every record that helps the consumers correlate duplicates, and every consumer can behave idempotently with respect to the processed records, the entire processing graph will exhibit exactly-once behaviour in a limited sense. Specifically, the aspects of record processing that relate to databases and idempotent API calls are covered. Conversely, message queues and Kafka topics that don't provide application-level deduplication are excluded from the exactly-once guarantee; duplicates will occur, and it is something we have come to accept.

Idempotence requires collaboration not only from the consuming process, but from all downstream resources that said process manipulates.

Presumably, message-oriented middleware is a transport mechanism that facilitates the distribution of messages between endpoints or the replication of events, whichever paradigm one chooses to accept; it is not the definitive source of truth, in that it does not attempt to replace a database. It would be unacceptable for a database to return multiple rows for the same entity. But Kafka, being part of the transport apparatus, somehow gets away with this — relying on the application to perform deduplication.

What if Kafka was used as primary storage — the proverbial 'source of truth'; for example, acting in the role of an event store in an event sourcing system? It would hardly be acceptable to have two records representing the same logical event. This is not to say that a consumer could not be designed to cope with duplicates, but it would be fair to assume that duplicates would be undesirable in most cases — irritating, to put it mildly.

The solution — transactions

The transactional messaging capability introduced in release 0.11.0.0 strengthens Kafka's delivery semantics. Namely, it introduces limited *atomicity*, *consistency* and *isolation* guarantees on top of the existing *durability* pledge, for a combined *ACID* experience that is characteristic of relational databases (and occasionally of NoSQL). Specifically —

- **Atomicity** — ensures that, for a group of records published within an encompassing transaction scope, either all records are *visibly* persisted to their respective logs, or none are persisted. In other words, the transaction either succeeds or fails as a whole. The logs, in this case, are not limited to a single partition; transactions may span multiple topics and partitions.

- **Consistency** — a logical corollary of atomicity, consistency ensures that the cluster transitions from one valid state to another, without violating any invariants in between. In our context, consistency speaks to the correlation between records on input and output topics of a processing stage; an input record must result in an output record once consumed, or it must not be consumed at all.
- **Isolation** — ensures that concurrent execution of transactions leaves the system in the same state that would have been obtained if the transactions were executed sequentially. In simple terms, the effects of a transaction cannot be externally visible until it commits. (In reality, the visibility of in-flight transactions depends on the configured isolation level of the consumer.)

Transactions under the hood

Role of the transaction coordinator

Chapter 10: Client Configuration had briefly touched on some of the inner workings of idempotent producers. At the heart of the implementation is a unique *producer ID* (PID) that is assigned by a *transaction coordinator* for the duration of the producer's session. A transaction coordinator is a module running within a broker process, servicing idempotent and transactional producers — akin to the group coordinator used by consumers to manage group membership. The transaction coordinator is responsible for issuing PIDs and managing transaction state. The producer starts by issuing an `InitProducerId` request to the appropriate transaction coordinator (there may be multiple coordinators); the latter replies with a unique PID that is valid for the duration of the producer's session.

Transactional messaging builds upon this infrastructure by increasing the lifetime of a producer's PID such that it survives a single producer session. This is achieved by specifying an optional `transactional.id` property on the producer. If the property is not set, the producer will be issued a new PID on each request; otherwise, the transaction coordinator will associate the issued PID with the transactional ID and maintain this association for the duration specified in the `transactional.id.expiration.ms` broker property, which defaults to `604800000` (one week). The *transactional ID* ⟶ *PID* mapping includes the epoch corresponding to the point when the association was last updated. The epoch acts as a fencing mechanism, blocking *zombie* processes that have been displaced by a newer PID assignment. (A zombie producer might attempt to use their PID to manipulate an in-flight transaction or initiate a new one; however, they will act using an older epoch number, which gives them away.) A `ProducerFencedException` is thrown when the producer attempts to manipulate a transaction that has been fenced off.

The load balancing of transaction coordinator duties is done by assigning the partitions of the internal `__transaction_state` topic to the brokers in the cluster, such that the assigned owner of any given partition becomes the notional coordinator for that partition's index. The identification of a suitable transaction coordinator is performed by hashing the transactional ID, modulo the partition index — arriving at the coordinator in charge of the partition. By piggybacking on an existing leadership election process, Kafka conveniently avoids yet another arbitration process to elect transaction coordinators. Also, the management of transactions is divided approximately equally across all the brokers in the cluster, which allows the transaction throughput to scale by increasing the number of brokers.

Producer API enhancements

Transactional messaging adds several methods to the `Producer` API, namely:

- `initTransactions()`: Initialises the transactional subsystem by obtaining the PID and epoch number, and finalising all prior transactions for the given transactional ID with the transaction coordinator. (And in doing so, fencing any zombies for the previous epoch.) This method should only be called once for a producer session, and must be called prior to any of the methods below. Once initialised, the producer will be transitioned to the `READY` state.
- `beginTransaction()`: Demarcates the start of transaction scope on the producer. Performs a series of local checks to ensure that the client is operating in transactional mode and transitions the producer to the `IN_TRANSACTION` state. This method must be invoked before any of the methods below.
- `sendOffsetsToTransaction()`: Incorporates the given set of consumer-side per-topic-partition offsets into the scope of the current transaction and forwards them to the group coordinator of the supplied consumer group. The offsets will come into effect when the transaction subsequently commits.
- `commitTransaction()`: Flushes any unsent records and commits the current transaction by instructing the transaction coordinator to do so. This places the producer into the `COMMITTING_TRANSACTION` state, after which it is not possible to invoke any other method. (It is possible to retry `commitTransaction()` in this state.)
- `abortTransaction()`: Discards any pending records for the current transaction and instructs the transaction coordinator to abort the transaction. This places the producer into the `ABORTING_TRANSACTION` state, from which point no other transactional methods may be invoked (except for retrying `abortTransaction()`).

Transactional messaging in Kafka is largely a producer-side concern. This makes sense, as *transactions control the atomicity of write operations*. The reader might recall from Chapter 3: Architecture and Core Concepts, Kafka employs a recursive approach to managing committed offsets, utilising the internal `__consumer_offsets` topic to persist and track offsets for consumer groups. Because the committing of an offset is modelled as a write, it is necessary for consumers to piggyback on an existing transaction scope managed by a producer. In other words, for a typical processing stage

that comprises both a consumer client and a producer client, the consumer must forward its offsets to the producer, rather than use its native `commitSync()` or `commitAsync()` API. On top of this, offset auto-commit must be disabled on the consumer; *the use of transactions demands manual offset committing*. There is one other aspect of transactional behaviour that must be accounted for — *isolation* — which will be covered shortly.

With transactions enabled, each iteration of a typical *consume-transform-produce* loop resembles the following:

```
producer.beginTransaction();
try {
  // process records ...
  producer.send(...);
  // ...
  producer.sendOffsetsToTransaction(...);
  producer.commitTransaction();
} catch (KafkaException e) {
  producer.abortTransaction();
  throw e;
}
```

Assigning a transactional ID

And now we arrive at the crux of the challenge — the choice of the transactional ID. This is widely considered as among the most confusing and perplexing aspects of transaction management in Kafka. Overwhelmingly, the questions in various forums that relate to transactions are posed around the transactional ID.

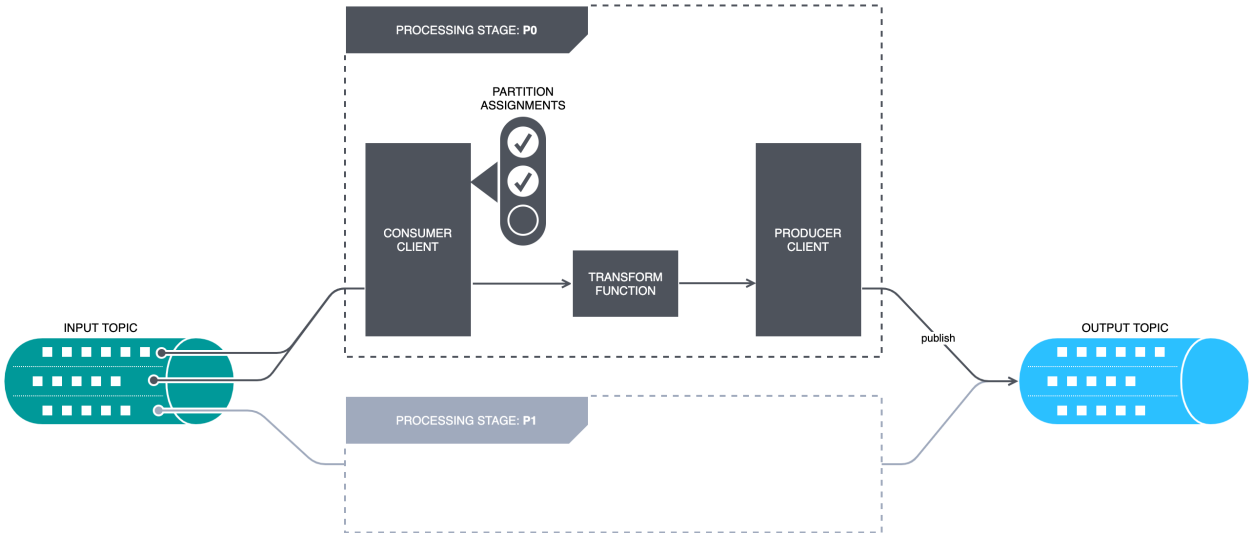
The reason this is so hard is that the transactional ID must survive producer sessions. It is also used as a fencing mechanism, which means that there must only be one (legitimate) producer using the transactional ID at any given time. This is further complicated by the fact that the number of processes in a stage is not a constant, and may scale in and out depending on load, as well as factors outside of our control, such as process restarts, network partitions, and so on.

Before revealing the recommended transactional ID assignment scheme, it is instructive to explore a handful of options to appreciate the complexity of the problem.

Starting with the simplest — a common transactional ID shared by all processes. We can dismiss this option hastily, as it creates an irreconcilable fencing issue for all but the most recent processes that present the shared transactional ID. As the assignment of PIDs is associated with an epoch number, the last producer to call `initTransactions()` will acquire the highest epoch number — turning its peers into zombies (from the perspective of the transaction coordinator).

Crossing to the opposite extreme — a random transactional ID with sufficient collision-resistance to guarantee its uniqueness. One might use UUIDs (versions 3, 4 and 5) or some other unique quantity.

This clearly avoids the issue of fencing legitimate peer processes, but in doing so it effectively disables zombie fencing altogether. Consider the implications by reverting to our earlier example of a simple processing stage, surrounded by a pair of input and output topics. Assume two processes *P0* and *P1* contend for the assignment of three partitions in the input topic *I0*, *I1* and *I2* — transforming the input and publishing records to some partitions in the output topic. The output partitions, as will shortly become apparent, are irrelevant — it is only the input partitions that matter for fencing.



Simple stream processing example

Initially, *P0* is assigned *I0* and *I1*, and *P1* is assigned *I2*. Both processes are operating with random transactional IDs. At some point, an issue on *P1* blocks its progress through *I2*, eventually resulting in failure detection on the group coordinator, followed by partition reassignment. Thereafter, *P0* becomes the assignee of all partitions in the input topic; *P1* is presumed dead.

What if *P1* had in-flight transactions at the time of reassignment? The transaction coordinator will provide for up to the value of `transactional.id.expiration.ms` to honour a pending transaction. This setting defaults to `3600000` (one hour). During this time, producers will be allowed to write to the affected partitions, but no record that appears after the partial transaction on the affected partitions will be delivered to consumers operating under the `read_committed` assurance level. That amount of downtime on the consumer is generally unacceptable in most stream processing applications. Of course, the expiration time could be wound down from its default value, but one must be careful not to interrupt the normal operation of transactional producers by setting an overly aggressive expiration time. At any rate, some appreciable amount of downtime cannot be avoided.

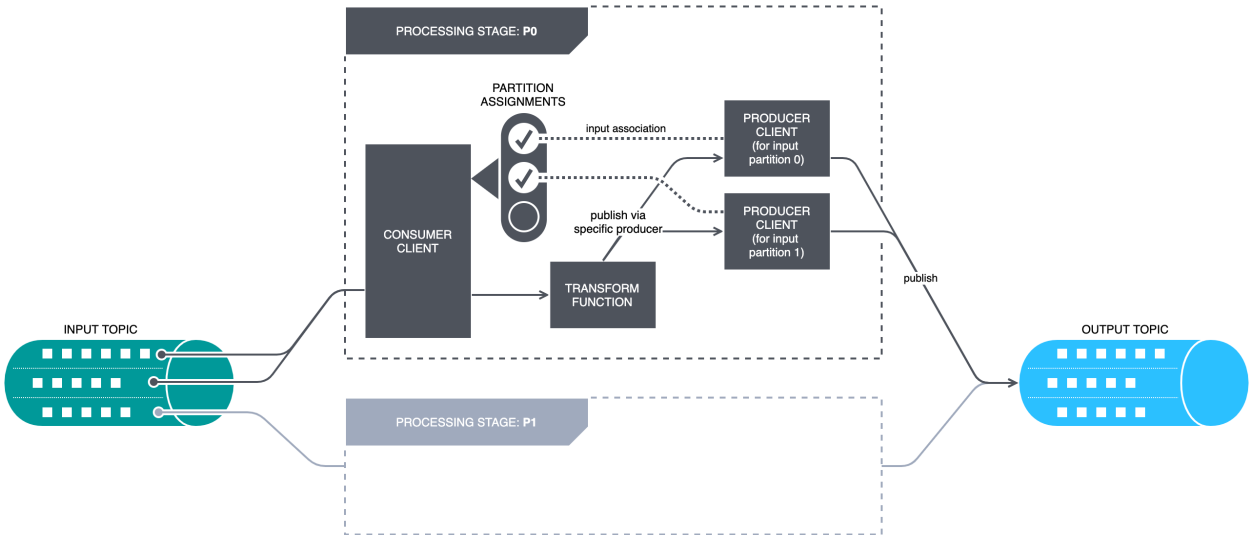
Another problem with this approach is what happens on *P1* if it suddenly resumes operating within the transaction expiry period. Being excluded from its consumer group, it will not be able to consume further records, but any in-flight records (returned from a previous poll) are still fair game from its perspective. It may attempt to publish more records on the output topic within the existing transaction scope — an operation that will be accepted by the transaction coordinator (which knows nothing about the consumer-side status of the process). In the worst case, the output topic will contain duplicates that have been emitted by the two processes, corresponding to the same input

record.

We can try our hand at various other transactional assignment schemes, but the result will invariably be the same — either undue fencing of legitimate producers or bitter defeat at the hands of zombies.

The very point of fencing is to eliminate these sorts of scenarios, but for fencing to function correctly, the producer's session must somehow relate back to the input source. This is where a confession must be made: the issue isn't just with the transactional ID assignment scheme, but with the internal architecture of the processing stage. In the original example, we used a pair of clients — a consumer and a producer — in a back-to-back arrangement. A process would read from any of the assigned partitions in the input set, via a single consumer instance. It would then publish a record (or multiple records) to an arbitrary output partition via single, long-lived producer instance.

Before continuing with the transactional ID conundrum, we need to rework the internal architecture of the processing stage. Let the singleton consumer be, but replace the singleton producer with a collection of producers — one for each assigned partition in the input set. The assignment of partitions is subject to change, which can be accommodated by registering a `ConsumerRebalanceListener`. When a partition is assigned, create and initialise a corresponding producer instance; when a partition is revoked, dispose of the producer instance. When publishing an output record, always use the producer instance that corresponds to the partition index of the input record. The diagram below illustrates this internal topological arrangement.



Pinned producers

The number of partitions in the input topic(s) might be in the hundreds or thousands; spawning a producer instance for each conceivable topic-partition pair is wasteful, entailing a potentially significant connection establishment overhead, in addition to utilising memory (for buffers), file handles and OS-level threads. As such, reducing the producer clients to the set of assigned partitions is the preferred approach. Furthermore, the producer instances may be lazily initialised — amortising the overhead from the point of partition assignment to the points of the first read from each input topic-partition.

Now for the reveal: the transactional ID for each producer instance is derived by concatenating the corresponding input topic and partition index pair.

To understand why this approach works, consider the earlier example with *P0* and *P1*, where *I2* was transferred from *P1* to *P0* as a result of a perceived failure. For simplicity, assume the input and output topics are named `tx-input` and `tx-output`, respectively. Under the revised architecture, *P0* starts with one consumer and two producers with transactional IDs `tx-input-0` and `tx-input-1`. *P1* has one transactional producer — `tx-input-2`. Upon reassignment, *P0* has `tx-input-0`, `tx-input-1` and `tx-input-2`. *P1 thinks* it has `tx-input-2`.

When *P0* acquires *I2* and instantiates a producer with the transactional ID `tx-input-2`, the call to `initTransactions()` will result in the finalisation of any pending transaction state with the transaction coordinator. The coordinator will either commit or roll back the transaction, depending on the state of its transaction log, recorded in the internal `__transaction_state` topic. If the outgoing producer was able to fire off a commit request prior to revocation, the transaction will be committed by writing `COMMITTED` control messages to the affected partitions in `tx-output`. Conversely, if the outgoing producer aborted the transaction or otherwise failed to explicitly end the transaction prior to revocation, the transaction will be forcibly rolled back by writing `ABORTED` control messages to the affected partitions. At any rate, any pending transaction will be finalised before *P0* is permitted to publish new records via its `tx-input-2` producer. As part of this ceremony, the epoch number is incremented.

When a transaction is rolled back, Kafka does not delete records from the affected partitions. Doing so would hardly sit well with the notion of an append-only ledger. Instead, by writing an `ABORTED` control message to each of the affected partitions, the coordinator signals to a transactional consumer that the records should not be delivered to the application.

In the meantime, *P1* might attempt to complete its transaction, acting under the `tx-input-2` transactional ID. It may add more records to the transaction scope, which is accompanied by an `AddPartitionsToTxnRequest` to the transaction coordinator. Alternatively, it may attempt to commit the transaction by sending an `EndTxnRequest`. Both requests include a mandatory epoch number. Because the initialisation of the producer on *P0* had incremented the epoch number on the transaction coordinator, all further actions under the lapsed epoch number will be disallowed by the coordinator.

To summarise: by pinning a producer client instance to the input topic-partition, we are capturing the causality among input and output records within the identity of the producer — its PID, by way of a derived transactional ID. As the partition assignment changes on the group coordinator, the causal relationship is carried forward to the new assignee; the outgoing assignee will fail if it attempts to publish a record under the same identity.

Transactional consumers

It was stated earlier that transactional messaging is largely a producer-side concern. Largely, but not wholly — consumers still have a role to play in implementing the *isolation* property of transactions.

Specifically, when records are published within transaction scope, these records are persisted to their target partitions even before the transaction commits (or aborts, for that matter). A transaction might span multiple records across disparate topics. For transactions to retain the isolation property, it is essential that the consumer ecosystem collaborates with the producer and the transaction coordinator. This is accomplished by terminating a sequence of records that form part of a transaction by a control marker — for every partition that is featured within the transaction scope.

All consumers above version 0.11.0.0 understand the notion of control markers; however, the way a consumer reacts to a control marker varies depending on the consumer's `isolation.level` setting. The default isolation level is `read_uncommitted` — meaning the consumer will disregard the control markers and deliver the records as-is to the polling application. This includes records that are part of in-flight transactions as well as those records that were part of an aborted transaction.

To enable transactional semantics on a consumer, the `isolation.level` must be set to `read_committed`. Within this mode, the behaviour of the producer changes with respect to the end offset. Normally, the end offsets are equivalent to the high-water mark — the offset immediately following the last successfully replicated record. A transactional consumer replaces its notion of end offsets with the *Last Stable Offset* (LSO) — the minimum of the high-water mark and the smallest offset of any open transaction. Under the constraint of the LSO, a producer will not be allowed to enter a region in the log that contains an open transaction — not until that transaction commits or aborts. In the former, the contents of the transaction will be delivered in the result of `Consumer.poll()`. In the latter, the records will be silently discarded.

Having acquired an understanding of how consumers are bound in their progression through the assigned partitions, it is easy to see why zombie fencing is an essential element of transactional messaging. Without fencing measures in place, a producer stuck in an open transaction will impede the advancement of the LSO for up to `transactional.id.expiration.ms` — impacting all downstream consumers running with `isolation.level=read_committed`.

Simple stream processing example

Having set the theoretical foundations for transactional messaging, it is time to implement the above scenario. To get started, we need a pair of topics — `tx-input` and `tx-output`:

```
$KAFKA_HOME/bin/kafka-topics.sh --bootstrap-server localhost:9092 \
    --create --topic tx-input --partitions 3 --replication-factor 1


$KAFKA_HOME/bin/kafka-topics.sh --bootstrap-server localhost:9092 \
    --create --topic tx-output --partitions 3 --replication-factor 1
```

The replication factor was kept to the allowed minimum to support our single-node test broker. Three partitions are used on either side to demonstrate the affinity between producer instances and

the input topic-partition. For simplicity, we are not using authentication or authorization in this example.

The complete source code listing for this example is available at github.com/ekoutanov/effectivekafka[47] in the `src/main/java/effectivekafka/transaction` directory.

The listing below depicts a simple transformation stage that takes an integer as input and squares it before publishing the resulting value in an output record.

```
import static java.lang.System.*;

import java.time.*;
import java.util.*;

import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.*;
import org.apache.kafka.common.serialization.*;

public final class TransformStage {
  public static void main(String[] args) {
    final var inputTopic = "tx-input";
    final var outputTopic = "tx-output";
    final var groupId = "transform-stage";

    final Map<String, Object> producerBaseConfig =
        Map.of(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9092",
               ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
               StringSerializer.class.getName(),
               ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
               IntegerSerializer.class.getName(),
               ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,
               true);

    final Map<String, Object> consumerConfig =
        Map.of(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9092",
               ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
               StringDeserializer.class.getName(),
```

[47] https://github.com/ekoutanov/effectivekafka/tree/master/src/main/java/effectivekafka/transaction

```
            ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            IntegerDeserializer.class.getName(),
            ConsumerConfig.GROUP_ID_CONFIG,
            groupId,
            ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
            "earliest",
            ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
            false);

final var producers = new PinnedProducers(producerBaseConfig);

try (var consumer =
     new KafkaConsumer<String, Integer>(consumerConfig)) {
  consumer.subscribe(Set.of(inputTopic),
                     producers.rebalanceListener());

  while (true) {
    final var inRecs = consumer.poll(Duration.ofMillis(100));

    // read the records, transforming their values
    for (var inRec : inRecs) {
      final var inKey = inRec.key();
      final var inValue = inRec.value();
      out.format("Got record with key %s, value %d%n",
                 inKey, inValue);

      // prepare the output record
      final var outValue = inValue * inValue;
      final var outRec =
          new ProducerRecord<>(outputTopic, inKey, outValue);
      final var topicPartition =
          new TopicPartition(inRec.topic(), inRec.partition());

      // acquire producer for the input topic-partition
      final var producer = producers.get(topicPartition);

      // transactionally publish record and commit input offsets
      producer.beginTransaction();
      try {
        producer.send(outRec);
        final var nextOffset =
            new OffsetAndMetadata(inRec.offset() + 1);
        final var offsets = Map.of(topicPartition, nextOffset);
```

```
            producer.sendOffsetsToTransaction(offsets, groupId);
            producer.commitTransaction();
          } catch (KafkaException e) {
            producer.abortTransaction();
            throw e;
          }
        }
      }
    }

    /**
     *  Mapping of producers to input topic-partitions.
     */
    private static class PinnedProducers {
      final Map<String, Object> baseConfig;

      final Map<String, Producer<String, Integer>>
          producers = new HashMap<>();

      PinnedProducers(Map<String, Object> baseConfig) {
        this.baseConfig = baseConfig;
      }

      ConsumerRebalanceListener rebalanceListener() {
        return new ConsumerRebalanceListener() {
          @Override
          public void onPartitionsRevoked
              (Collection<TopicPartition> partitions) {
            for (var topicPartition : partitions) {
              out.format("Revoked %s%n", topicPartition);
              disposeProducer(getTransactionalId(topicPartition));
            }
          }

          @Override
          public void onPartitionsAssigned
              (Collection<TopicPartition> partitions) {
            for (var topicPartition : partitions) {
              out.format("Assigned %s%n", topicPartition);
              createProducer(getTransactionalId(topicPartition));
            }
          }
```

```
      };
    }

    Producer<String, Integer> get(TopicPartition topicPartition) {
      final var transactionalId = getTransactionalId(topicPartition);
      final var producer = producers.get(transactionalId);
      Objects.requireNonNull(producer,
                             "No such producer: " + transactionalId);
      return producer;
    }

    String getTransactionalId(TopicPartition topicPartition) {
      return topicPartition.topic()
          + "-" + topicPartition.partition();
    }

    void createProducer(String transactionalId) {
      if (producers.containsKey(transactionalId))
        throw new IllegalStateException("Producer already exists: "
            + transactionalId);

      final var config = new HashMap<>(baseConfig);
      config.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG,
                 transactionalId);
      final var producer =
          new KafkaProducer<String, Integer>(config);
      producers.put(transactionalId, producer);
      producer.initTransactions();
    }

    void disposeProducer(String transactionalId) {
      final var producer = producers.remove(transactionalId);
      Objects.requireNonNull(producer,
                             "No such producer: " + transactionalId);
      producer.close();
    }
  }
}
```

The example employs a single consumer client and multiple producers. The producer configuration is named `producerBaseConfig` because each producer's configuration will be slightly different — having a distinct `transactional.id` setting. The management of producers is conveniently delegated to the `PinnedProducers` class, which keeps a mapping of transactional IDs to producer instances. The

producer lifecycle is managed by exposing a `ConsumerRebalanceListener`, which spawns a producer upon partition assignment and disposes of the producer upon revocation. The `transactional.id` value is derived by concatenating the input topic name with the partition index, delimited by a hyphen character. For example, `tx-input-2` for partition 2 in the topic named `tx-input`.

Returning to the `main()` method, we have a classic *consume-process-publish* loop. Having consumed a batch of records, the application iterates over the batch and acquires a corresponding pinned producer instance for each record. A transaction is started for each input record, containing the output `send()` call as well as the committing of offsets via the producer. Note, all write operations *must* go via the producer API.

Before we can see this example in action, we need a way of generating input records and viewing the output. This is taken care of by a pair of classes — `InputStage` and `OutputStage`:

```
import static java.lang.System.*;

import java.util.*;

import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.*;

public final class InputStage {
  public static void main(String[] args)
      throws InterruptedException {
    final var topic = "tx-input";

    final Map<String, Object> config =
        Map.of(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9092",
               ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
               StringSerializer.class.getName(),
               ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
               IntegerSerializer.class.getName(),
               ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,
               true);

    try (var producer = new KafkaProducer<String, Integer>(config)) {
      while (true) {
        final var key = new Date().toString();
        final var value = (int) (Math.random() * 1000);

        out.format("Publishing record with key %s, value %d%n",
                   key, value);
        producer.send(new ProducerRecord<>(topic, key, value));
```

```
        Thread.sleep(500);
      }
    }
  }
}
```

```
import static java.lang.System.*;

import java.time.*;
import java.util.*;

import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.serialization.*;

public final class OutputStage {
  public static void main(String[] args) {
    final var topic = "tx-output";
    final var groupId = "output-stage";

    final Map<String, Object> config =
        Map.of(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9092",
               ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
               StringDeserializer.class.getName(),
               ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
               IntegerDeserializer.class.getName(),
               ConsumerConfig.GROUP_ID_CONFIG,
               groupId,
               ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
               "earliest",
               ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
               false,
               ConsumerConfig.ISOLATION_LEVEL_CONFIG,
               "read_committed");

    try (var consumer = new KafkaConsumer<String, Integer>(config)) {
      consumer.subscribe(Set.of(topic));

      while (true) {
        final var records = consumer.poll(Duration.ofMillis(100));
        for (var record : records) {
```

```
            out.format("Got record with key %s, value %d%n",
                       record.key(), record.value());
          }
          consumer.commitAsync();
        }
      }
    }
}
```

Run all three, starting them in any order. Each will do their thing; the `OutputStage` application will display the resulting records. The `TransformStage` app has additional logging in place to show the assignment and revocation of partitions. The logging isn't overly interesting when running a single transform instance, but when multiple processes are launched, it clearly highlights where the revocations of partitions on one consumer correspond to the assignment of partitions on the other.

Limitations

There are several limitations of transactional messaging in Kafka which the reader should be mindful of.

Bound to Kafka resources

Transactional messaging is implemented using a proprietary Kafka protocol that is not interoperable with other persistence systems and messaging middleware. Kafka does not support standard transaction APIs such as XA and JTA.

Cannot span producers

Transactions cannot be used to span multiple producer instances. Transactions are forwarded to specific transaction coordinators, which are assigned the role of managing transactional IDs by way of a hash function. Two producers with different transactional IDs may map to two different coordinators, making cross-producer transactions intractable within the constraints of the current design.

Cannot span clusters

Where the producer and consumer sides of a processing stage connect to different clusters, consumer-side offsets cannot be committed by relaying them through the transaction coordinator, as the latter resides in a different cluster.

May be partially observed by a consumer

As transactions can span multiple partitions, it is possible for any given consumer operating within a consumer group to only witness a subset of the records emitted within a broader transaction, being oblivious to the effects of the transaction that do not affect its assigned partitions. Because partition assignment is balanced among members of a consumer group, the records of one transaction may be subdivided among multiple consumers in a group.

Transactions may straddle multiple log segments in a partition. When the lapsed log segments are eventually deleted as a result of retention constraints, it may appear that parts of the transaction have vanished. In practice, this problem only impacts consumers who have accumulated significant lag, or are starting to read a topic from the beginning.

Records published within a transaction are not accorded any special treatment from the perspective of compaction. A background compaction process may void individual records, leaving a partial set. To be fair, the behaviour of transactional records with respect to retention and compaction is not a limitation of transactions as such, but more of a caveat that warrants awareness.

Incomplete exactly-once semantics

The biggest limitation of transactional messaging with respect to exactly-once processing is that it does nothing to prevent an input record from being handled twice; if the process suffers from a failure mid-stream, an alternate process will take over and may need to deal with partially processed records. This is straightforward if the processing stage embodies a pure function; in other words, it is both deterministic and has no state of its own. On the other hand, if the processing stage must write to a database or invoke a downstream service, prior side effects must be taken into account.

Are transactions over-hyped?

For all the proverbial 'tyre pumping' of the transactional messaging capability, one must question the added complexity of dealing with transactions and pinning producers, in terms of benefits that it provides. Assuming all persistent side effects are idempotent, a competently written *consume-transform-produce* loop provides the requisite exactly-once semantics end-to-end. Of course, this assumes we don't care about duplicates in Kafka, and one might go as far as arguing that the at-least-once tenets of event stream processing already imply some degree of robustness *vis à vis* duplicates, provided there is a reliable way of identifying them.

Transactional messaging eliminates duplicate records in event stream processing graphs. This may be useful when applications rely on Kafka as a primary event store and where dealing with duplicate records may be non-trivial at the application level. The complexity of managing pinned consumers may be encapsulated within a dedicated messaging layer, or a framework. For example, the *Kafka Streams* client library can transparently deal with transactions, relieving the application from having to manage consumers and producers, demarcate transactions and commit offsets. Kafka Streams is a

good fit for a broad range of map-reduce style and simple windowed operations, but understandably, it might not solve all your stream processing needs.

> When the fruits of KIP-98[a] were released to the general public, many an opportunity was exploited to publicise the implications of transparently facilitating exactly-once delivery guarantees at the middleware level. Suggestions were made that the new capability solves one of the most difficult problems in distributed computing, and one that was previously considered impossible.
>
> It was stated in Chapter 6: Design Considerations, that *exactly-once semantics are not possible at the middleware layer without tight-knit collaboration with the application.* This axiom applies to the general case, where record processing entails side effects. It is important to appreciate that this statement holds in spite of Kafka's achievements in the area of transactional messaging. The latter solves a limiting case of the exactly-once problem; specifically, it eliminates duplicate records in a graph of event stream processing stages, where each stage comprises exclusively of pure functions. Once again, there is no 'silver bullet'.
>
> It should be acknowledged that the *exactly-once impossibility* dictum does not take anything away from Kafka; the release of transactional messaging is nonetheless useful in a limited sense. Where the application domain does not fit entirely into the limiting case for which transactional messaging holds, the reader ought to take their own measures in ensuring idempotence across all affected resources.
>
> [a] https://cwiki.apache.org/confluence/x/ASD8Aw

For the majority of event-driven applications, the most useful and practical aspect of transactional messaging is the idempotence guarantee on the producer. This feature utilises the same underlying PID concept and transactional infrastructure, ensuring that records do no arrive out-of-order on the broker within the timeframe allowed by `delivery.timeout.ms`.

We have just emerged from one of the most taxing topics in all of Kafka. Transactional messaging was introduced in version 0.11.0.0, bringing about capabilities to support a combination of producer idempotence and end-to-end exactly-once delivery guarantees.

Transactional messaging improves upon Kafka *durability* guarantee, adding *atomicity*, *consistency* and *isolation* with respect to the production and consumption of records in a graph of stream processing applications. When operating within transaction scope, processing stages can avoid duplicate records in the output topics — ensuring a one-to-one correspondence between the set of input records and the output set.

The challenges of transactional messaging are largely concentrated in the mapping of stage inputs to outputs, by way of a stable transactional ID. The latter identifies a logical producer session spanning multiple physical producer instances. As partitions are reassigned among consumer processes, the transactional ID follows the partition assignment, resuming the processing of records and fencing

prior activity that might still be in effect. Implementing transactional stages requires the application to maintain multiple producer instances that are pinned to the origin of the record, managing the lifecycle of the producers in response to changes in partition assignments.

In its leaner guise, the underlying transactional infrastructure can be used to ensure that the publishing of any given record is idempotent insofar as the queued record will appear exactly-once on its target partition, and in the order prescribed by the producer. (A guarantee that holds within the extent of the delivery timeout, but could be contravened if the record is queued a second time.) This feature has an immediate and practical benefit, and is recommended to be enabled on all producers — not just stream processing graphs.