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:

```java
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.DefaultPartitione` 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.

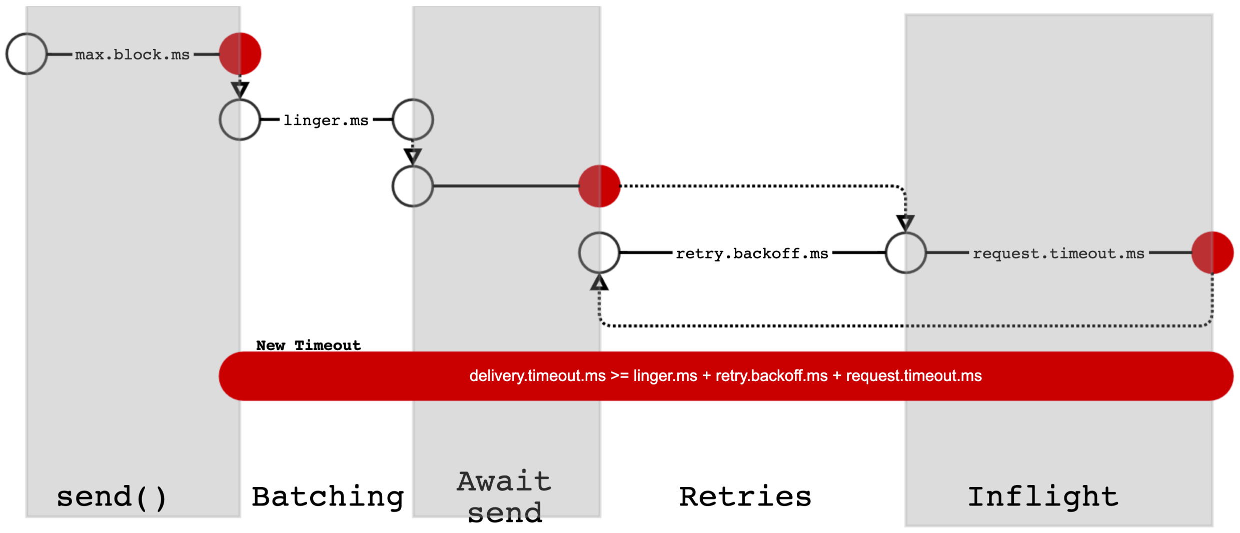
The producer may report failure to send a record earlier than this time if either an unrecoverable error is encountered, the retries have been exhausted, or the record is added to a batch which reached an earlier delivery expiration deadline. The value of this property should be greater than or equal 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.