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.

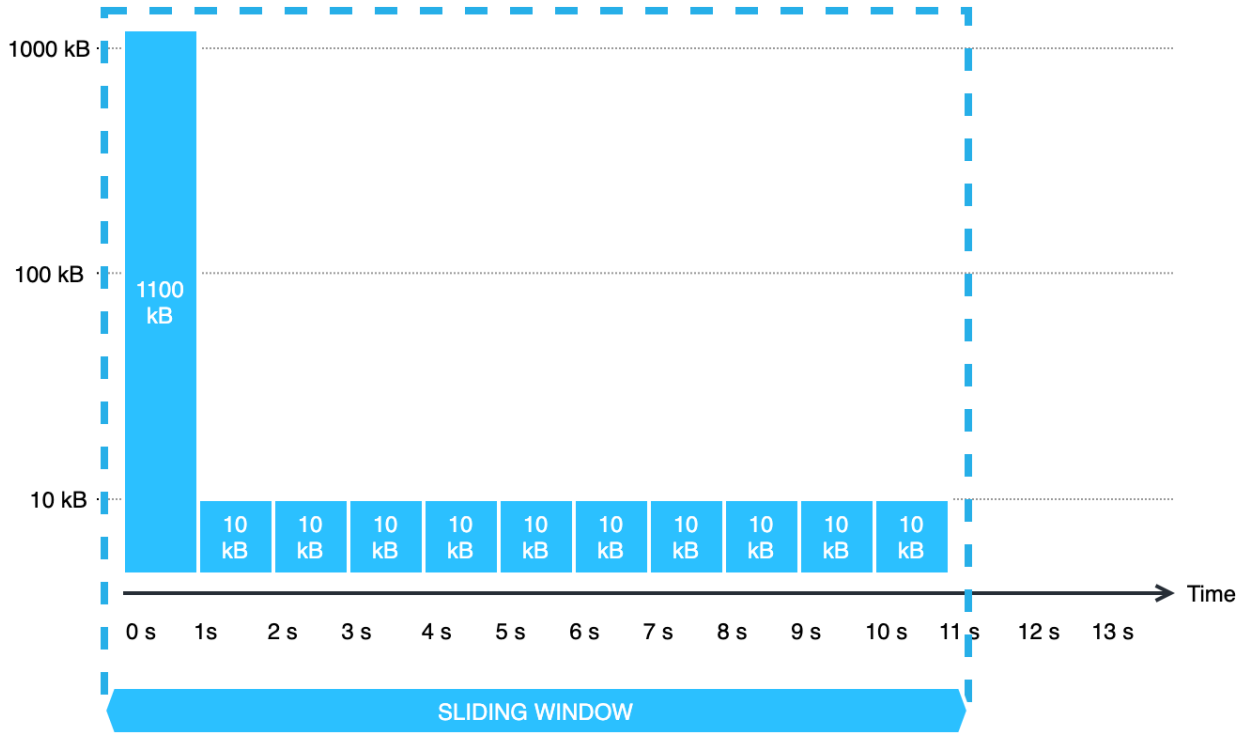
At this point, the client is being delayed by a small amount of time, as the extent of overproduction is quite minimal. Delayed does not mean stopped; the client will transmit the next batch after the broker's response is received. The batch size is given by the `batch.size` client property, which defaults 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.

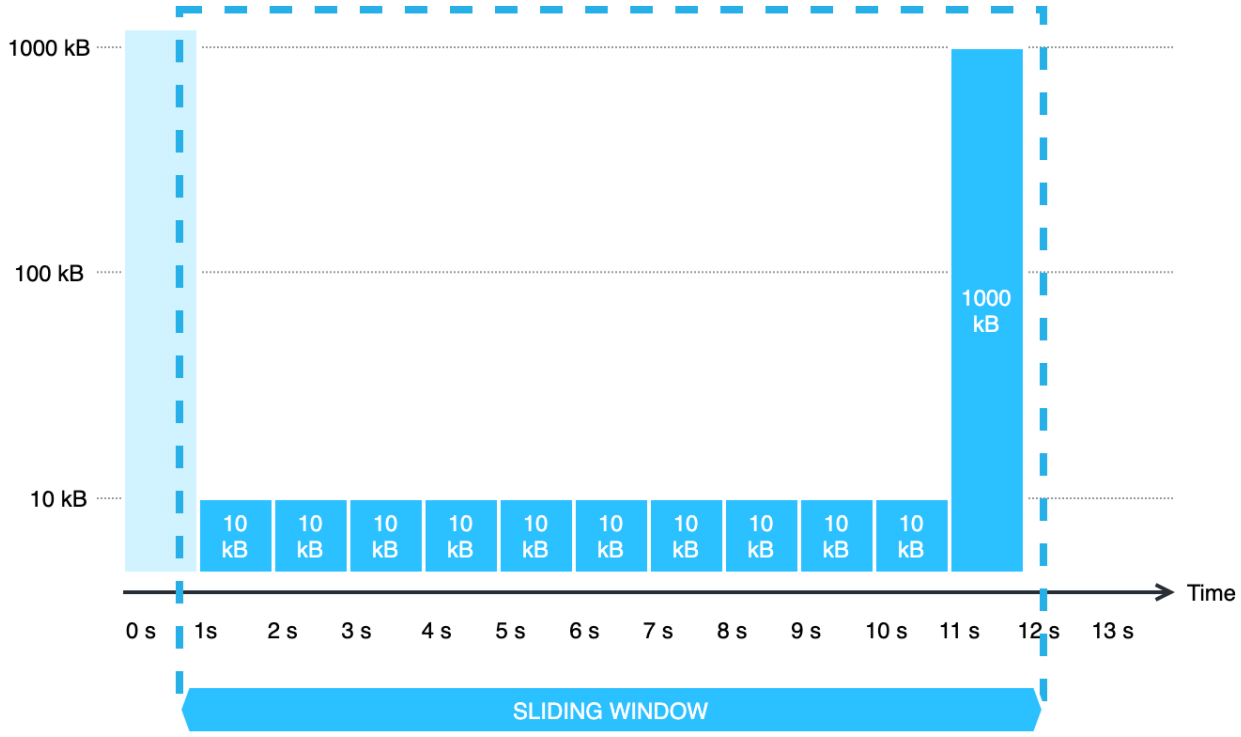


Sliding window — first cycle

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.

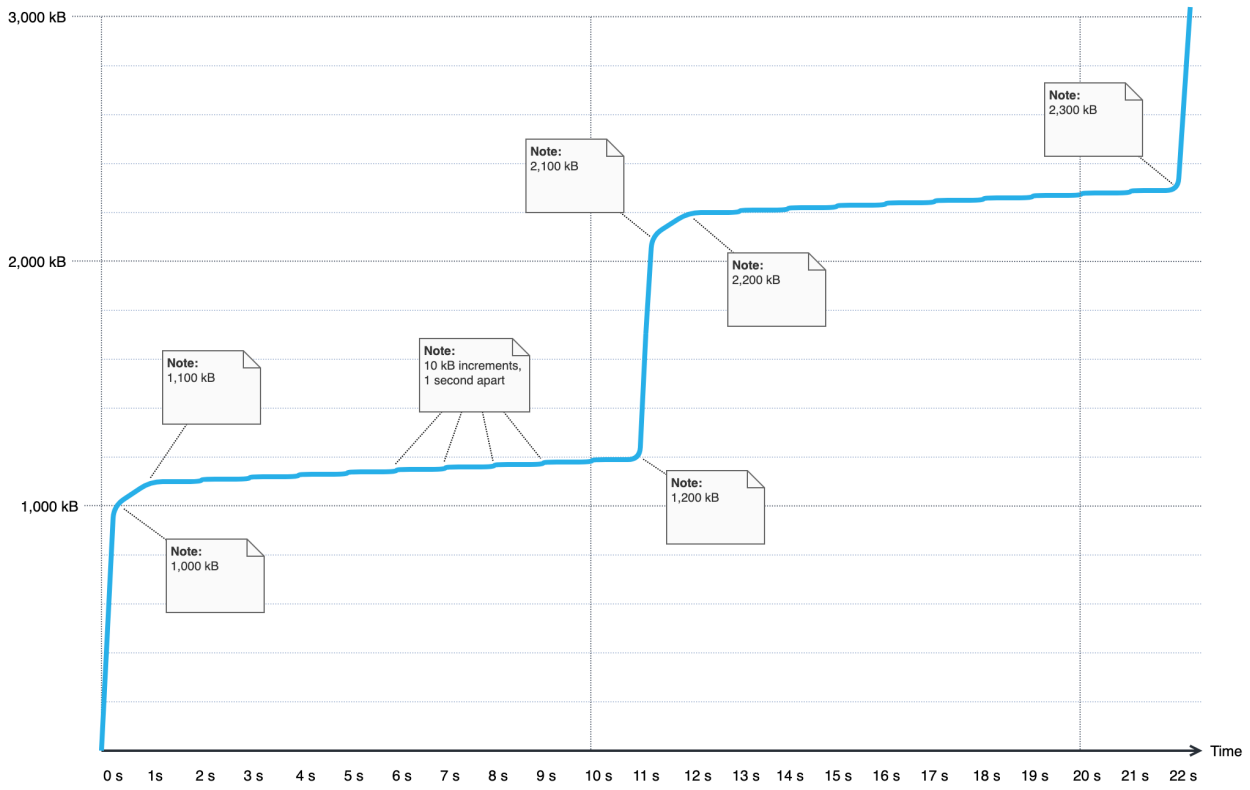
Upon the conclusion of the tenth second, we will have one sampling period with a high average utilisation value, followed by ten periods with relatively low utilisation values. When the 12th sampling period commences on the 11th second, the sliding window will have advanced to the point where the first 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.