

10

Protecting Kafka

This chapters covers

- Security basics and related terminology
- SSL between a cluster and clients
- Access control lists (ACLs)
- Network bandwidth and request rate quotas to limit demands on resources

This chapter focuses on keeping our data secured so that only those that need to read from or write to it have access. Because security is a huge area to cover, in this chapter, we will talk about some basic concepts to get a general background on the options we have in Kafka. Our goal in this chapter is not to set up security, but to learn some different options that you can talk with your security team on researching in the future and get familiar with the concepts. This will not be a complete guide to security in general, but sets the foundation for you. We will discuss practical actions you can take in your own setup, and we will look at the client impact, as well as brokers and ZooKeeper, to make our cluster more secure.

Your data might not need those protections we discuss, but knowing your data is key to deciding if you need the trade-offs of managing access. If you are handling

anything related to personal information or financial data, like date of birth or credit card numbers, then you will likely want to look at most of the security options discussed in this chapter. However, if you are only handling generic information such as marketing campaigns, or you are not tracking anything of a secure nature, then you might not need this protection. If this is the case, then your cluster would not need to introduce features like SSL. We start with an example of fictional data that we want to protect.

Let's imagine that we have a goal to find the location of a prize by taking part in a treasure hunt. As a competition-wide exercise, we have two teams, and we do not want the other team to access our own team's work. Starting out, each team picks their own topic names and shares that name with their team members only. (Without knowing which topic name to write to and read from, your data is out of the view of the other team.) Each team begins by sending their clues to what they assume is their own *private* topic. Over time, members of the teams might start to wonder about the progress of the other team and whether they have any clues that the other team doesn't. This is when the trouble starts. Figure 10.1 shows the topic setup for Team Clueful and Team Clueless.

At this point, nothing but team behavior is stopping the teams from reading each topic. Each team can read and write to each topic.

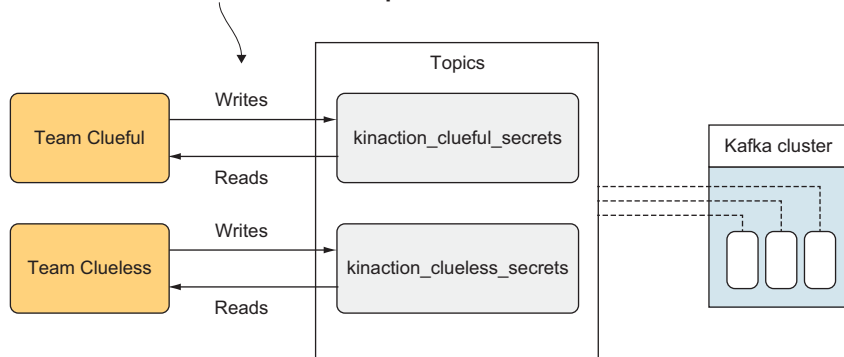


Figure 10.1 Treasure hunt topics

One tech-savvy competitor, who coincidentally has used Kafka before, reaches for his command line tools to find the topics (the other team's as well as his own). After getting a list of topics, the competitor now knows his rival's topic. Let's say that this team member of Team Clueless looks at Team Clueful's topic, `--topic kinaction_clueful_secrets`. With great happiness, all it took was a consumer console command to list all the data that Team Clueful has been working on so far in the competition! But the bad actor does not stop there.

In order to throw Team Clueful off the trail, the actor also writes false information into the channel. Now Team Clueful has bad data in their topic, which is hindering their clue-solving progress! Because they are not sure who really wrote the messages on their topic, Team Clueful now has to determine which are the false messages and,

in doing so, will lose valuable time that they could be using to work on figuring out the grand-prize location.

How could we avoid the situation Team Clueful finds itself in? Is there a way that only those clients that have permission would be able to read from or write to our topics? There are two parts to our solution. The first part is how to encrypt our data. The next is how to find out who a person is in our system; not only who they are, but also making sure that the claimed identity of the user is verified. Once we verify a user, we need to know what they are permitted to do in our system. We will dive deeper into these topics as we look at a few solutions provided with Kafka.

10.1 Security basics

In regard to computer application security, you will likely encounter encryption, authentication, and authorization at some point in your work. Let's take a closer look at this terminology (see <http://mng.bz/o802> for more detail of the following terms if needed).

Encryption does not mean that others might not see your messages, but that if they do, they will not be able to derive the original content that you are protecting. Many people will think of how they are encouraged to use a site that is secure (HTTPS) for online shopping on a Wi-Fi® network. Later, we are going to enable SSL (Secure Sockets Layer) for our communication, not between a website and our computer, but between our clients and brokers! As a general note, as we work through this chapter, the label “SSL” is the property name you will see in our examples and explanations even though TLS is the newer protocol version [1].

Moving along, let's talk about *authentication*. To verify the identity of a user or an application, we need to have a way to authenticate that user: authentication is the process of proving that a user or application is indeed who they claim to be. If you wanted to sign up for a library card, for example, does the library issue a card to anyone without making sure the user is who they say they are? In most cases, the library would confirm the person's name and address with something like a government-issued ID and a utility bill. This process is intended to ensure that someone cannot easily claim another identity to use for their own purposes. If someone claims your identity to borrow books and never returns them, sending the fines your way, you can easily see a drawback of not confirming the user's claim.

Authorization, on the other hand, focuses on what the user can do. Continuing with our library example, a card issued to an adult might provide different permissions than if it was given to a user considered to be a child. And access to online publications might be limited to only terminals inside the library for each cardholder.

10.1.1 Encryption with SSL

So far, all of our brokers in this book have supported plaintext [1]. In effect, there has been no authentication or encryption over the network. Knowing this, it might make sense to review one of the broker server configuration values. If you look at any of your current `server.properties` files (see appendix A for your setup location of the

config/server0.properties file, for example), you will find an entry like `listeners = PLAINTEXT:localhost://:9092`. That listener is, in effect, providing a mapping of a protocol to a specific port on the broker. Because brokers support multiple ports, this entry allows us to keep the `PLAINTEXT` port up and running, so we can test adding SSL or other protocols on a different port. Having two ports helps to make our transition smoother when we shift away from plaintext [2]. Figure 10.2 shows an example of using plaintext versus SSL.

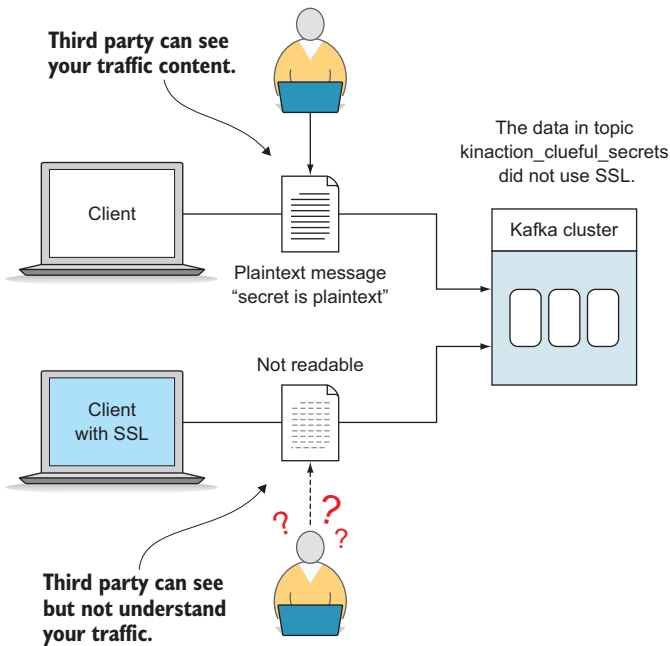


Figure 10.2 Plaintext vs. SSL

At this point, we are starting with a cluster without any security baked in. (Luckily, we can add various pieces to our cluster as we harden it against other teams.) Setting up SSL between the brokers in our cluster and our clients is one place to start [1]. No extra servers or directories are needed. No client coding changes are required, as the changes are configuration driven.

We don't know how advanced other users are when it comes to listening to our traffic on the same Wi-Fi network with security tools, so we know that we might not want to send plaintext from our brokers to our clients. Although the setup in the following section is needed for Kafka security, readers who have set up SSL or HTTPS in the past (and especially with Java) will find this approach similar to other client/server trust arrangements.

10.1.2 *SSL between brokers and clients*

In our previous examples of writing clients and connecting to Kafka, we have not used SSL for connections. However, now we are going to look at turning it on for the

communication between our clients and our cluster to encrypt our network traffic with SSL. Let's walk through the process and see what we are going to need to accomplish in order to get our cluster updated with this feature.

NOTE The commands in this chapter are specific and will not work the same on all operating systems (or even across different server domain names listed for broker setup) without modification. The important thing is to follow along with the general concepts. Moreover, other tools (like OpenSSL®) can be switched out, so your setup and commands might be different. But once you get the concepts, head to Confluent's site at <http://mng.bz/nrza> for even more resources and guides. Confluent's documents that provided direction for any examples are referenced throughout this chapter and should be referenced to help you actually implement the topics we only cover at a high level in order to introduce the following concepts.

WARNING A security professional should be consulted for the correct way to set up your own environment. Our commands are meant as a guide for getting familiar and for learning, not as a production level of security. This is not a complete guide. Use it at your own risk!

One of our first steps is to create a key and certificate for our brokers [3]. Because you should already have Java on your machine, one option is to use the `keytool` utility, which is part of the Java installation. The `keytool` application manages a keystore of keys and trusted certificates [4]. The important part to note is the *storage*. In this chapter, the term *broker0* is included in some filenames to identify one specific broker, not one that is meant for every broker. It might be good to think of a keystore as a database where our JVM programs can look up this information for our processes when needed [4]. At this point, we are also going to generate a key for our brokers as in the following listing [3]. Note that `manning.com` is used as an example in the following listings and is not intended to be used for readers following along.

Listing 10.1 SSL key generation for a broker

```
keytool -genkey -noprompt \  
-alias localhost \  
-dname "CN=ka.manning.com,OU=TEST,O=TREASURE,L=Bend,S=Or,C=US" \  
-keystore kafka.broker0.keystore.jks \  
-keyalg RSA \  
-storepass changeTreasure \  
-keypass changeTreasure \  
-validity 999
```

← Names the keystore that holds our newly generated key

← Uses a password so that the store cannot be changed without it

After running this command, we will have created a new key and stored it in the keystore file `kafka.broker0.keystore.jks`. Because we have a key that (in a way) identifies our broker, we need something to signal that we don't have just any certificate issued by a random user. One way to verify our certificates is by signing them with a CA (certificate

authority). You might have heard of CAs offered by Let's Encrypt® (<https://letsencrypt.org/>) or GoDaddy® (<https://www.godaddy.com/>), to name a few sources. The role of a CA is to act as a trusted authority that certifies the ownership and identity of a public key [3]. In our examples, however, we are going to be our own CA to avoid any need of verifying our identity by a third party. Our next step is to create our own CA, as the following listing shows [3].

Listing 10.2 Creating our own certificate authority

```
openssl req -new -x509 \
  -keyout cakey.crt -out ca.crt \
  -days 999 \
  -subj '/CN=localhost/OU=TEST/O=TREASURE/L=Bend/S=Or/C=US' \
  -passin pass:changeTreasure -passout pass:changeTreasure
```

Creates a new CA, then produces
key and certificate files

This generated CA is now something that we want to let our clients know that they should trust. Similar to the term *keystore*, we will use a truststore to hold this new information [3].

Because we generated our CA in listing 10.2, we can use it to sign our certificates for our brokers that we have already made. First, we export the certificate that we generated in listing 10.2 for each broker from the keystore, sign that with our new CA, and then import both the CA certificate and signed certificate back into the keystore [3]. Confluent also provides a shell script that can be used to help automate similar commands (see <http://mng.bz/v497>) [3]. Check out the rest of our commands in the source code for the book in the section for this chapter.

NOTE While running the commands in these listings, your operating system or tool version may have a different prompt than that passed. It will likely have a user prompt appear after running your command. Our examples try to avoid these prompts.

As part of our changes, we also need to update the `server.properties` configuration file on each broker, as the following listing shows [3]. Note that this listing only shows `broker0` and only part of the file.

Listing 10.3 Broker server properties changes

```
...
listeners=PLAINTEXT://localhost:9092,
➡ SSL://localhost:9093
ssl.truststore.location=
➡ /opt/kafkainaction/private/kafka
➡ .broker0.truststore.jks
ssl.truststore.password=changeTreasure
ssl.keystore.location=
➡ /opt/kafkainaction/kafka.broker0.keystore.jks
ssl.keystore.password=changeTreasure
ssl.key.password=changeTreasure
...
```

Adds the SSL broker port, leaving
the older PLAINTEXT port

Provides the truststore location
and password for our broker

Provides the keystore location
and password for our broker

Changes are also needed for our clients. For example, we set the value `security.protocol=SSL`, as well as the truststore location and password in a file called `custom-ssl.properties`. This helps set the protocol used for SSL as well as points to our truststore [3].

While testing these changes, we can also have multiple listeners set up for our broker. This also helps clients migrate over time, as both ports can serve traffic before we drop the older `PLAINTEXT` port for our clients [3]. The `kinaction-ssl.properties` file helps our clients provide the information needed to interact with the broker that is now becoming more secured!

Listing 10.4 Using SSL configuration for command line clients

```
bin/kafka-console-producer.sh --bootstrap-server localhost:9093 \
--topic kinaction_test_ssl \
--producer.config kinaction-ssl.properties
bin/kafka-console-consumer.sh --bootstrap-server localhost:9093 \
--topic kinaction_test_ssl \
--consumer.config kinaction-ssl.properties
```

Let's our producer know about the SSL details

Uses our SSL configuration for consumers

One of the nicest features is that we can use the same configuration for both producers and consumers. As you look at the contents of this configuration file, one issue that might spring to mind is the use of passwords in these files. The most straightforward option is to make sure that you are aware of the permissions around this file. Limiting the ability to read as well as the ownership of the file is important to note *before* placing this configuration on your filesystem. As always, consult your security experts for better options that might be available for your environment.

10.1.3 SSL between brokers

Another detail to research since we also have our brokers talking to each other is that we might want to decide if we need to use SSL for those interactions. We can use `security.inter.broker.protocol = SSL` in the server properties if we do *not* want to continue using plaintext for communications between brokers and consider a port change as well. More details can be found at <http://mng.bz/4KBw> [5].

10.2 Kerberos and the Simple Authentication and Security Layer (SASL)

If you have a security team that already has a Kerberos server, you likely have some security experts to ask for help. When we first started working with Kafka, it was with a part of a suite of big data tools that mostly used Kerberos. Kerberos is often found in organizations as a method to provide single sign-on (SSO) that is secure.

If you have a Kerberos server set up already, you need to work with a user with access to that Kerberos environment to create a principal for each broker and also for each user (or application ID) that will access the cluster. Because this setup might be too involved for local testing, follow along with this discussion to see the format of Java

Authentication and Authorization Service (JAAS) files, which is a common file type for brokers and clients. There are great resources at <http://mng.bz/QqxG> if you want to gain more details [6].

JAAS files, with keytab file information, help to provide Kafka with the principal and credentials that we will use. A *keytab* will likely be a separate file that has the principal and encrypted keys. We can use this file to authenticate to the Kafka brokers without requiring a password [7]. However, it is important to note that you need to treat your keytab file with the same security and care that you would for any credential.

To get our brokers set up, let's look at some server property changes we'll need to make and an example JAAS configuration. To start, each broker will need its own keytab file. Our JAAS file will help our brokers find the keytab's location on our server, as well as declare the principal to use [7]. The following listing shows an example JAAS file brokers would use on startup.

Listing 10.5 Broker SASL JAAS file

```
KafkaServer {
  ...
  keyTab="/opt/kafkainaction/kafka_server0.keytab"
  principal="kafka/kafka0.ka.manning.com@MANNING.COM";
};
```

← Sets up the Kafka broker JAAS file

We are going to add another port to test SASL_SSL before we remove the older ports [7]. The following listing shows this change. Depending on what port you used to connect to your brokers, the protocol is either PLAINTEXT, SSL, or SASL_SSL in this example.

Listing 10.6 Changing the broker SASL properties

```
listeners=PLAINTEXT://localhost:9092,SSL://localhost:9093,
➡ SASL_SSL://localhost:9094
```

← Adds the SASL_SSL broker port, leaving the older ports

The setup for a client is similar [7]. A JAAS file is needed, as the following listing shows.

Listing 10.7 Client SASL JAAS file

```
KafkaClient {
  ...
  keyTab="/opt/kafkainaction/kafkaclient.keytab"
  principal="kafkaclient@MANNING.COM";
};
```

← Adds the client SASL JAAS file entry

We also need to update client configuration for the SASL values [3]. The client file is similar to our kinaction-ssl.properties file used earlier, but this one defines the SASL_SSL protocol. After testing that things are not broken on port 9092 or 9093, we can

use our new configuration by validating the same result as before when we use our new SASL_SSL protocol.

10.3 Authorization in Kafka

Now that we have seen how to use authentication with Kafka, let's take a look at how we can start using that information to enable user access. For this discussion, we'll start with access control lists.

10.3.1 Access control lists (ACLs)

As a quick review, authorization is the process that controls what a user can do. One way to enable authorization is with access control lists (ACLs). Although most Linux users are familiar with permissions on a file they can control with a `chmod` command (such as read, write, and execute), one drawback is that the permissions might not be flexible enough for our needs. ACLs can provide permissions for multiple individuals and groups as well as more types of permissions, and they are often used when we need different levels of access for a shared folder [8]. One example is a permission to let a user edit a file but not allow the same user to delete it (delete is a separate permission altogether). Figure 10.3 shows Franz's access to the resources for our hypothetical team for our treasure hunt.

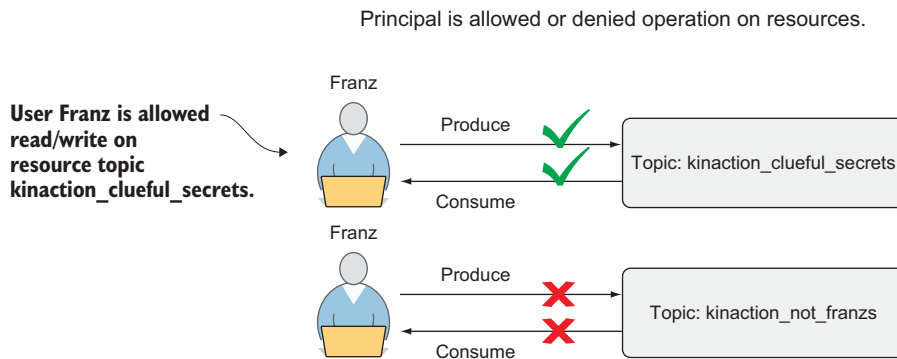


Figure 10.3 Access control lists (ACLs)

Kafka designed their authorizer to be pluggable, which allows users to make their own logic if desired [8]. Kafka has a `SimpleAclAuthorizer` class that we will use in our example.

Listing 10.8 shows adding the authorizer class and superuser Franz to the broker's `server.properties` file in order to use ACLs. An important item to note is that once we configure an authorizer, we need to set ACLs, or only those considered superusers will have access to any resources [8].

Listing 10.8 ACL authorizer and superusers

```
authorizer.class.name=
➡ kafka.security.auth.SimpleAclAuthorizer
super.users=User:Franz
```

Every broker configuration should include the SimpleAclAuthorizer.

Adds a superuser that can access all resources with or without ACLs

Let's see how to grant access to Team Clueful so that only that team produces and consumes from their own topic, `kinaction_clueful_secrets`. For brevity, we use two users in our example team, Franz and Hemingway. Because we have already created the keytabs for the users, we know the principal information that we need. As you may notice in the following listing, the operation `Read` allows consumers the ability to get data from the topic [8]. The second operation, `Write`, allows the same principals to produce data into the topic.

Listing 10.9 Kafka ACLs to read and write to a topic

```
bin/kafka-acls.sh --authorizer-properties \
  --bootstrap-server localhost:9094 --add \
  --allow-principal User:Franz \
  --allow-principal User:Hemingway \
  --operation Read --operation Write \
  --topic kinaction_clueful_secrets
```

Identifies two users to grant permissions

Allows the named principals to both read from and write to the specific topic

The `kafka-acls.sh` CLI tool is included with the other Kafka scripts in our installation and lets us add, delete, or list current ACLs [8].

10.3.2 Role-based access control (RBAC)

Role-based access control (RBAC) is an option that the Confluent Platform supports. RBAC is a way to control access based on roles [9]. Users are assigned to their role according to their needs (such as a job duty, for example). Instead of granting every user permissions, with RBAC, you manage the privileges assigned to predefined roles [9]. Figure 10.4 shows how adding a user to a role gives them a new permission assignment.

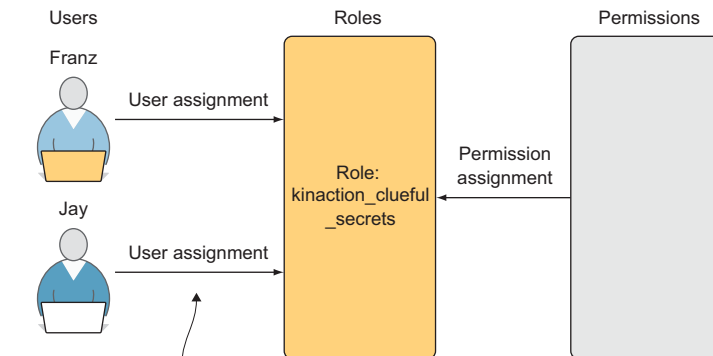


Figure 10.4
Role-based access control (RBAC)

User added. Nothing changes besides a user added to the role.

For our treasure hunting teams, it might make sense to have a specific role per team. This might mirror how a team from marketing would have a role versus a team from accounting. If some user changes departments, their role would be reassigned and not their individual permissions. Because this is a newer option, which may change as it matures and which is geared to the Confluent Platform environment, this is mentioned for awareness. We will not dig further into it here.

10.4 ZooKeeper

Part of securing Kafka is looking at how we can secure all parts of our cluster, including ZooKeeper. If we protect the brokers but not the system that holds that security-related data, it is possible for those with knowledge to update security values without much effort. To help protect our metadata, we will need to set the value `zookeeper.set.acl` to `true` per broker, as shown in the following listing [10].

Listing 10.10 ZooKeeper ACLs

```
zookeeper.set.acl=true
```

← Every broker configuration includes this ZooKeeper value.

10.4.1 Kerberos setup

Making sure that ZooKeeper works with Kerberos requires a variety of configuration changes. For one, in the `zookeeper.properties` configuration file, we want to add those values that let ZooKeeper know that SASL should be used for clients and which provider to use. Refer to <http://mng.bz/Xr0v> for more details if needed [10]. While we were busy looking at the other options for setup so far in this chapter, some users on our treasure hunt system were still up to no good. Let's see if we can dig into the subject of quotas to help with that.

10.5 Quotas

Let's say that some users of our web application don't have any issues with requesting data repeatedly. Although this is often a good thing for end users that want to use a service as much as they want without their progress being limited, our cluster may need some protection from users who might use that to their advantage. In our example, because we made it so the data was accessed by members of our team only, some users on the opposing team thought of a new way to prevent members of our team from working successfully. In effect, they are trying to use a distributed denial-of-service (DDoS) attack against our system [11]!

A targeted attack against our cluster can overwhelm our brokers and their surrounding infrastructure. In practice, the other team is requesting reads from our topics over and over while reading from the beginning of the topics each time they request data. We can use quotas to prevent this behavior. One detail that's important to know is that quotas are defined on a per-broker basis [11]. The cluster does not look across each broker to calculate a total, so a per-broker definition is needed. Figure 10.5 shows an example of using a request percentage quota.

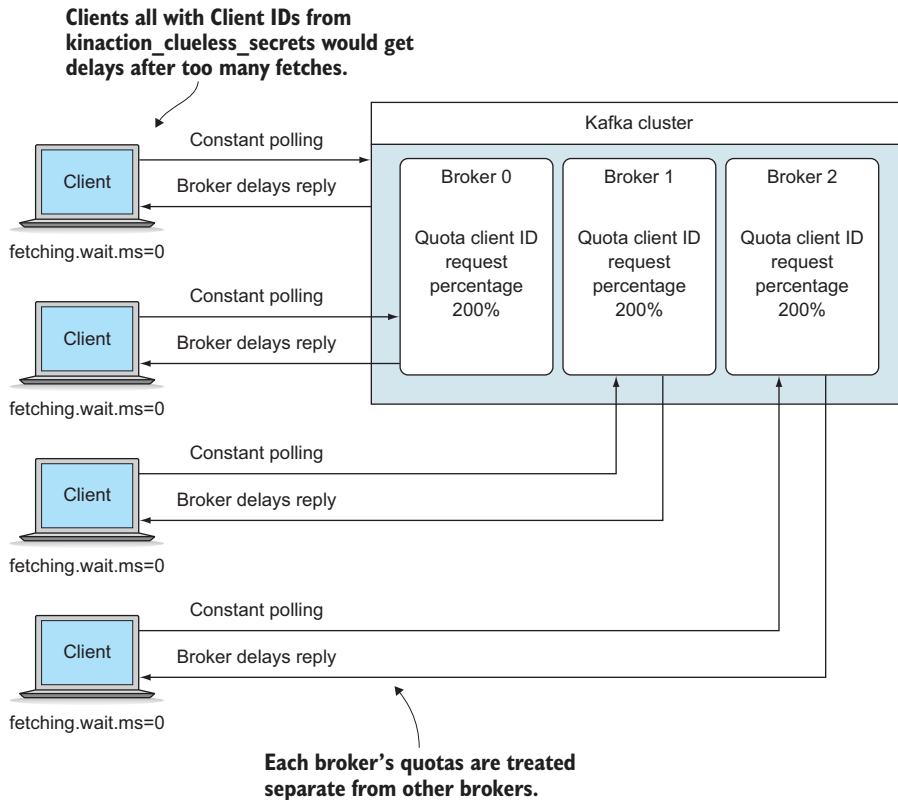


Figure 10.5 Quotas

To set our own custom quotas, we need to know how to identify *who* to limit and the *limit* we want to set. Whether we have security or not impacts what options we have for defining who we are limiting. Without security, we are able to use the `client.id` property. With security enabled, we can also add the user and any user and `client.id` combinations as well [11]. There are a couple of types of quotas that we can look at defining for our clients: network bandwidth and request rate quotas. Let's take a look at the network bandwidth option first.

10.5.1 Network bandwidth quota

Network bandwidth is measured by the number of bytes per second [12]. In our example, we want to make sure that each client is respecting the network and not flooding it to prevent others from using it. Each user in our competition uses a client ID that is specific to their team for any producer or consumer requests from their clients. In the following listing, we'll limit the clients using the client ID `kinaction_clueful` by setting a `producer_byte_rate` and a `consumer_byte_rate` [13].

Listing 10.11 Creating a network bandwidth quota for client `kinaction_clueful`

```
bin/kafka-configs.sh --bootstrap-server localhost:9094 --alter \
  --add-config 'producer_byte_rate=1048576,
  ➡ consumer_byte_rate=5242880' \
  --entity-type clients --entity-name kinaction_clueful
```

Names the entity for a client
with the client.id `kinaction_clueful`

Limits producers
to 1 MB per second
and consumers to
5 MB per second

We used the `add-config` parameter to set both the producer and consumer rate. The `entity-name` applies the rule to our specific `kinaction_clueful` clients. As is often the case, we might need to list our current quotas as well as delete them if they are no longer needed. All of these commands can be completed by sending different arguments to the `kafka-configs.sh` script, as the following listing shows [13].

Listing 10.12 Listing and deleting a quota for client `kinaction_clueful`

```
bin/kafka-configs.sh --bootstrap-server localhost:9094 \
  --describe \
  --entity-type clients --entity-name kinaction_clueful
```

Lists the existing configuration
of our client.id

```
bin/kafka-configs.sh --bootstrap-server localhost:9094 --alter \
  --delete-config
  ➡ 'producer_byte_rate,consumer_byte_rate' \
  --entity-type clients --entity-name kinaction_clueful
```

Uses delete-config to remove
those we just added

The `--describe` command helps us get a look at the existing configuration. We can then use that information to decide if we need to modify or even delete the configuration by using the `delete-config` parameter.

As we start to add quotas, we might end up with more than one quota applied to a client. We need to be aware of the precedence in which various quotas are applied. Although it might seem like the most restrictive setting (the lowest bytes allowed) would be the highest for quotas, that is not always the case. The following is the order in which quotas are applied with the highest precedence listed at the top [14]:

- User and `client.id`-provided quotas
- User quotas
- `client.id` quotas

For example, if a user named Franz has a user-quota limit of 10 MB and a `client.id` limit of 1 MB, the consumer he uses would be allowed 10 MB per second due to the user-defined quota having higher precedence.

10.5.2 Request rate quotas

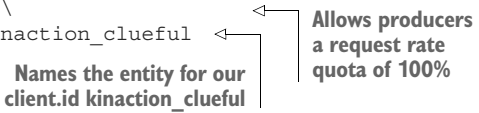
The other quota to examine is *request rate*. Why the need for a second quota? Although a DDoS attack is often thought of as a network issue, clients making lots of connections could still overwhelm the broker by making CPU-intensive requests. Consumer

clients that poll continuously with a setting of `fetch.max.wait.ms=0` are also a concern that can be addressed with request rate quotas, as shown in figure 10.5 [15].

To set this quota, we use the same entity types and `add-config` options as we did with our other quotas [13]. The biggest difference is setting the configuration for `request_percentage`. You'll find a formula that uses the number of I/O threads and the number of network threads at <http://mng.bz/J6Yz> [16]. In the following listing, we set a request percentage of 100 for our example [13].

Listing 10.13 Creating a network bandwidth quota for client `kinaction_clueful`

```
bin/kafka-configs.sh --bootstrap-server localhost:9094 --alter \
  --add-config 'request_percentage=100' \
  --entity-type clients --entity-name kinaction_clueful
```



Names the entity for our client.id `kinaction_clueful`

Allows producers a request rate quota of 100%

Using quotas is a good way to protect our cluster. Even better, it lets us react to clients that suddenly might start putting a strain on our brokers.

10.6 Data at rest

Another thing to consider is whether you need to encrypt the data that Kafka writes to disk. By default, Kafka does not encrypt the events it adds to its logs. There have been a couple of Kafka Improvement Proposals (KIPs) that have looked at this feature, but at the time of publication, you will still need to make sure you have a strategy that meets your requirements. Depending on your business needs, you might want to only encrypt specific topics or even specific topics with unique keys.

10.6.1 Managed options

If you use a managed option for your cluster, it might be best to check out what features the service provides. Amazon's Managed Streaming for Apache Kafka (<https://aws.amazon.com/msk/>) is one example of a cloud provider that handles a large part of your cluster management, including some security pieces. Having your brokers and ZooKeeper nodes updated with automatically deployed hardware patches and related upgrades addresses one major method of keeping issues at bay. The other benefit of these updates is that you are not providing access to your cluster for even more developers. Amazon MSK also provides encryption for your data and with TLS between various components of Kafka [17].

Additional management features that we covered in our examples in this chapter included the ability to use SSL between your clients and cluster and ACLs. Confluent Cloud (<https://www.confluent.io/confluent-cloud/>) also is an option that can be deployed across various public cloud offerings. Support for data encryption at rest and in motion as well as ACL support are also options that you should be aware of when matching your security requirements to the actual provider.

Sticking with the Confluent stack, Confluent Platform 5.3 has a commercial feature called *secret protection* (<http://mng.bz/yJYB>). When we looked at our SSL configuration files earlier, we stored plaintext passwords in certain files. However, secret protection is meant to address that issue by encrypting the secrets in the file and keeping exposed values out of files as well [18]. Because this is a commercial offering, we do not go into depth on how it works, but just be aware, there are options available.

Summary

- Plaintext, although fine for prototypes, needs to be evaluated before production usage.
- SSL (Secure Sockets Layer) can help protect your data between clients and brokers and even between brokers.
- You can use Kerberos to provide a principal identity, allowing you to use Kerberos environments that already exist in an infrastructure.
- Access control lists (ACLs) help define which users have specific operations granted. Role-based access control (RBAC) is also an option that the Confluent Platform supports. RBAC is a way to control access based on roles.
- Quotas can be used with network bandwidth and request rate limits to protect the available resources of a cluster. These quotas can be changed and fine-tuned to allow for normal workloads and peak demand over time.

References

- 1 “Encryption and Authentication with SSL.” Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/authentication_ssl.html (accessed June 10, 2020).
- 2 “Adding security to a running cluster.” Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/incremental-security-upgrade.html#adding-security-to-a-running-cluster> (accessed August 20, 2021).
- 3 “Security Tutorial.” Confluent documentation (n.d.). https://docs.confluent.io/platform/current/security/security_tutorial.html (accessed June 10, 2020).
- 4 `keytool`. Oracle Java documentation (n.d.). <https://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html> (accessed August 20, 2021).
- 5 “Documentation: Incorporating Security Features in a Running Cluster.” Apache Software Foundation (n.d.). http://kafka.apache.org/24/documentation.html#security_rolling_upgrade (accessed June 1, 2020).
- 6 V. A. Brennen. “An Overview of a Kerberos Infrastructure.” Kerberos Infrastructure HOWTO. <https://tldp.org/HOWTO/Kerberos-Infrastructure-HOWTO/overview.html> (accessed July, 22, 2021).
- 7 “Configuring GSSAP.” Confluent documentation (n.d.). https://docs.confluent.io/platform/current/kafka/authentication_sasl/authentication_sasl_gssapi.html (accessed June 10, 2020).

- 8 “Authorization using ACLs.” Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/authorization.html> (accessed June 10, 2020).
- 9 “Authorization using Role-Based Access.” Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/security/rbac/index.html> (accessed June 10, 2020).
- 10 “ZooKeeper Security.” Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/security/zk-security.html> (accessed June 10, 2020).
- 11 “Quotas.” Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/design.html#quotas> (accessed August 21, 2021).
- 12 “Network Bandwidth Quotas.” Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/design.html#network-bandwidth-quotas> (accessed August 21, 2021).
- 13 “Setting quotas.” Apache Software Foundation (n.d.). <https://kafka.apache.org/documentation/#quotas> (accessed June 15, 2020).
- 14 “Quota Configuration.” Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/design.html#quota-configuration> (accessed August 21, 2021).
- 15 KIP-124 “Request rate quotas.” Wiki for Apache Kafka. Apache Software Foundation (March 30, 2017). <https://cwiki.apache.org/confluence/display/KAFKA/KIP-124+-+Request+rate+quotas> (accessed June 1, 2020).
- 16 “Request Rate Quotas.” Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/kafka/design.html#request-rate-quotas> (accessed August 21, 2021).
- 17 “Amazon MSK features.” Amazon Managed Streaming for Apache Kafka (n.d.). <https://aws.amazon.com/msk/features/> (accessed July 23, 2021).
- 18 “Secrets Management.” Confluent documentation (n.d.). <https://docs.confluent.io/platform/current/security/secrets.html> (accessed August 21, 2021).