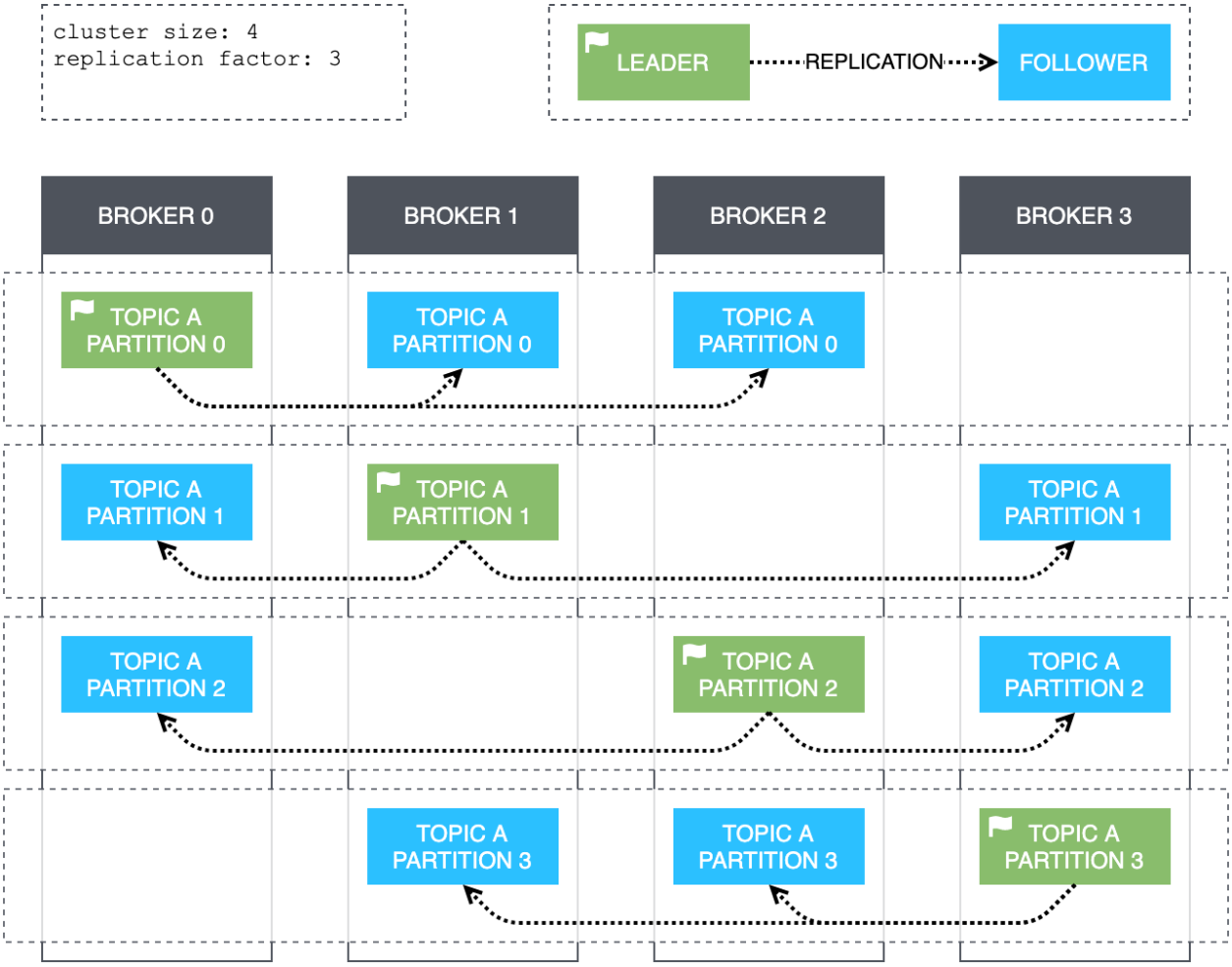
Chapter 8: Bootstrapping and Advertised Listeners

Who are these listeners? And what are they advertising?

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

A gentle introduction to bootstrapping

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



Topic replication

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

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

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

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

Taking advantage of DNS

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

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

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

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

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

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

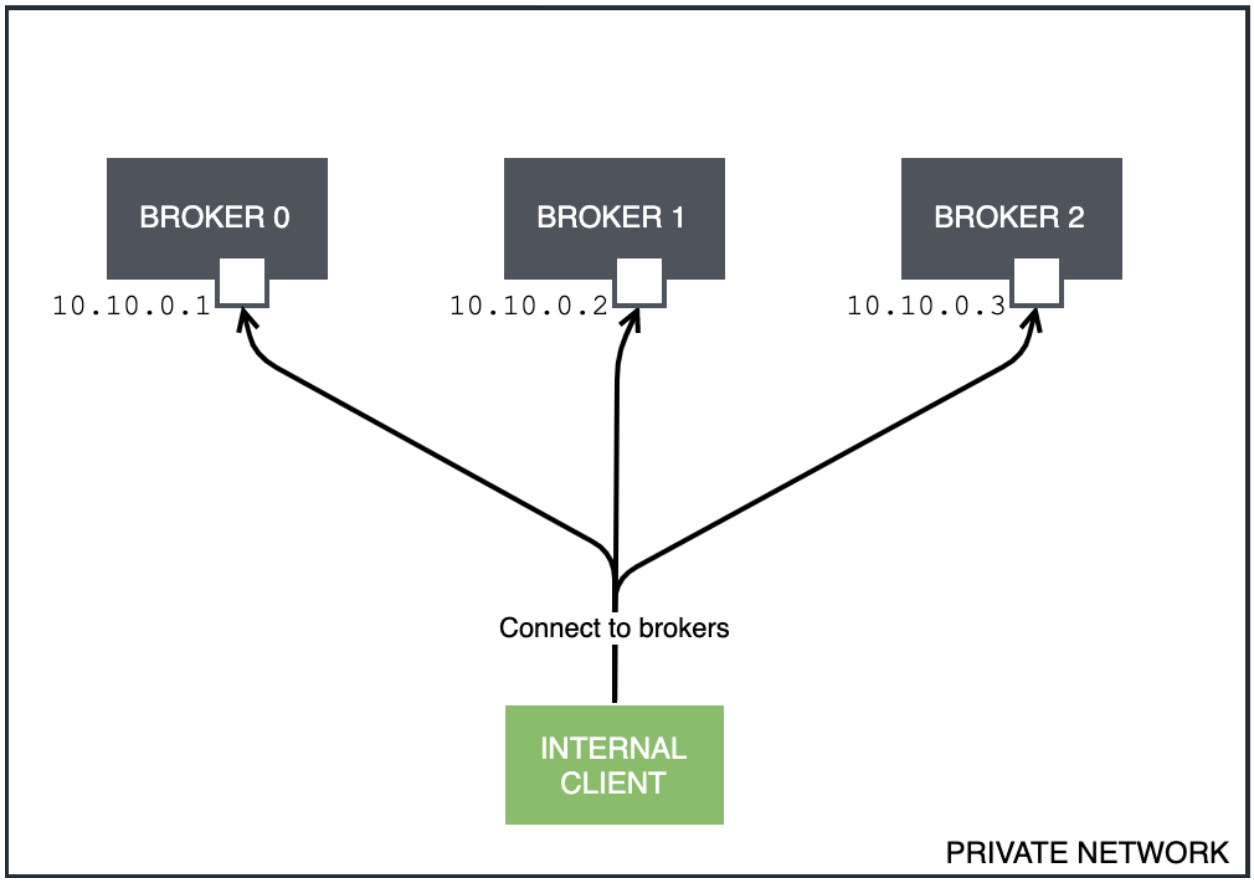
The advantage of this approach is that it does not require us to alter the bootstrap list when adding more fallback addresses. The list may be reduced to a single entry — for example, `broker.ext.prod.kafka.mycompany` — which will resolve to an arbitrary number of IP addresses, depending on how the DNS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

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

A simple scenario

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

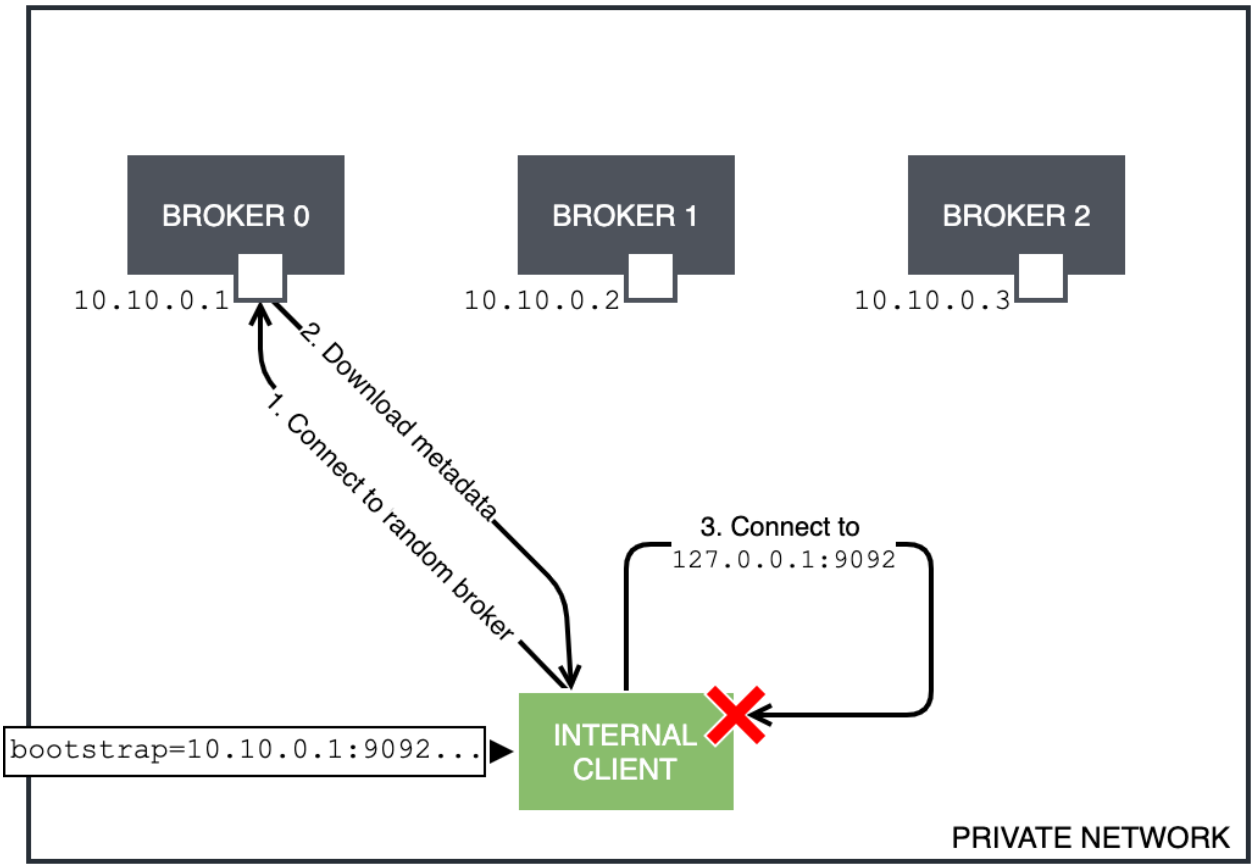
[10] https://cwiki.apache.org/confluence/display/KAFKA/KIP-302+-+Enable+Kafka+clients+to+use+all+DNS+resolved+IP+addresses



Connecting within a private network

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

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



Internal client looping back to itself

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

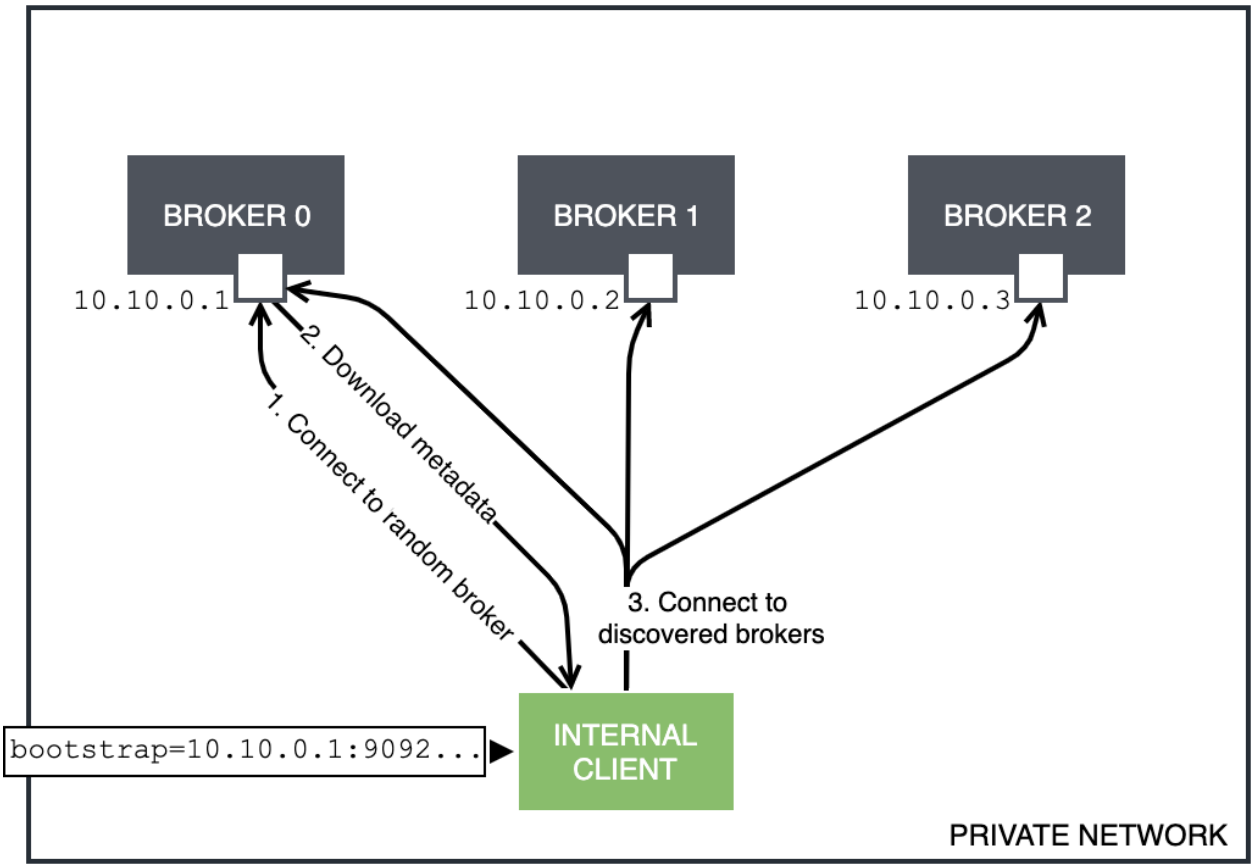
In addition to specifying a socket listener in `listeners`, you need to state how the listener is advertised to producer and consumer clients. This is done by appending an entry to `advertised.listeners`, in the form of `<listener protocol>://<advertised host name>:<advertised port>`. Returning to our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

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

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

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

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



Bootstrapping over an internal network

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

Kafdrop, showing broker metadata

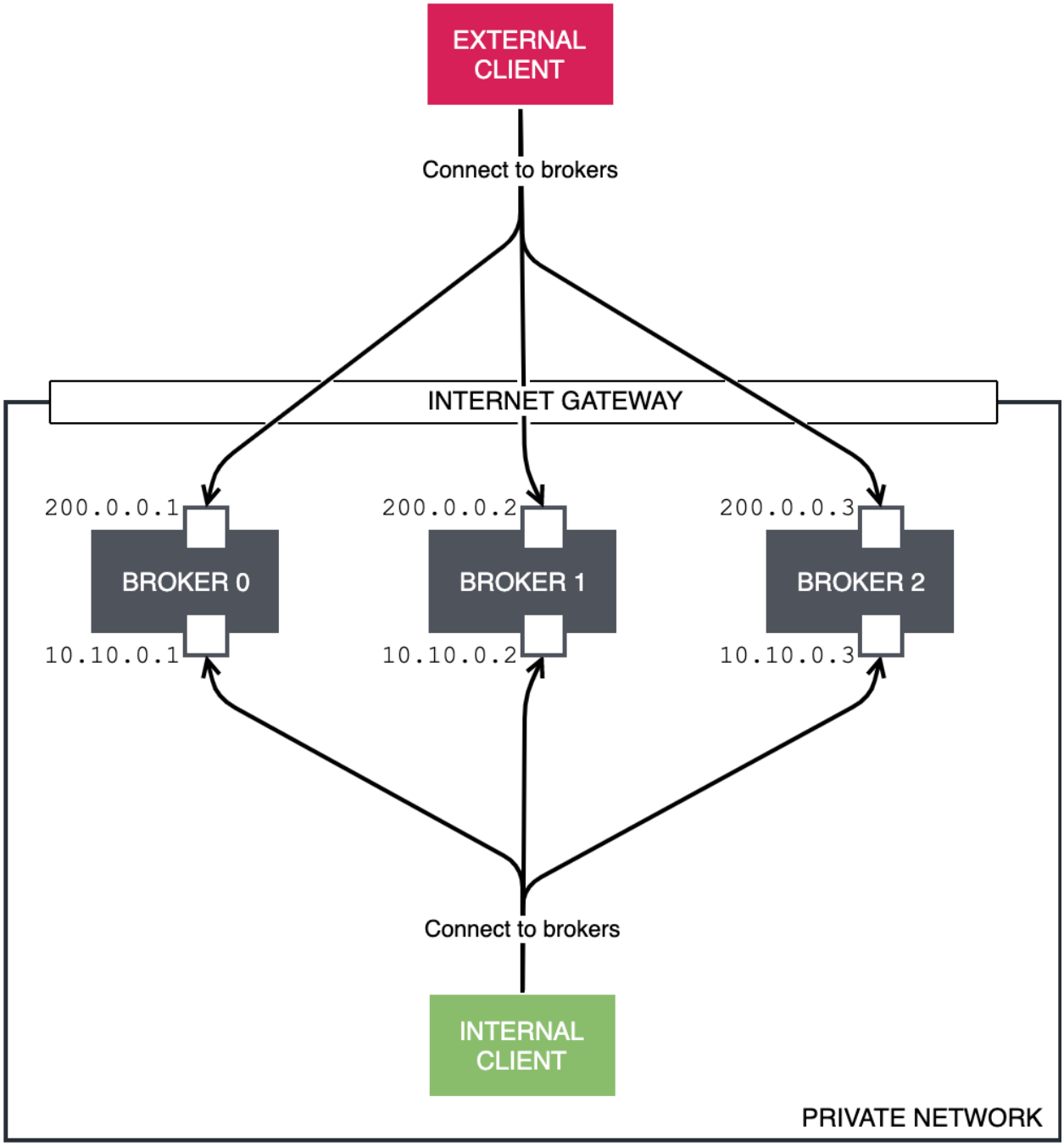
Earlier in this chapter, we touched upon the `client.dns.lookup` property and the effect of setting it to `use_all_dns_ips` — specifically, the Java client's ability to utilise all resolved DNS entries. When enabled, the advertised listeners will also be subjected to the same DNS treatment. If an advertised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

Multiple listeners

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

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

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



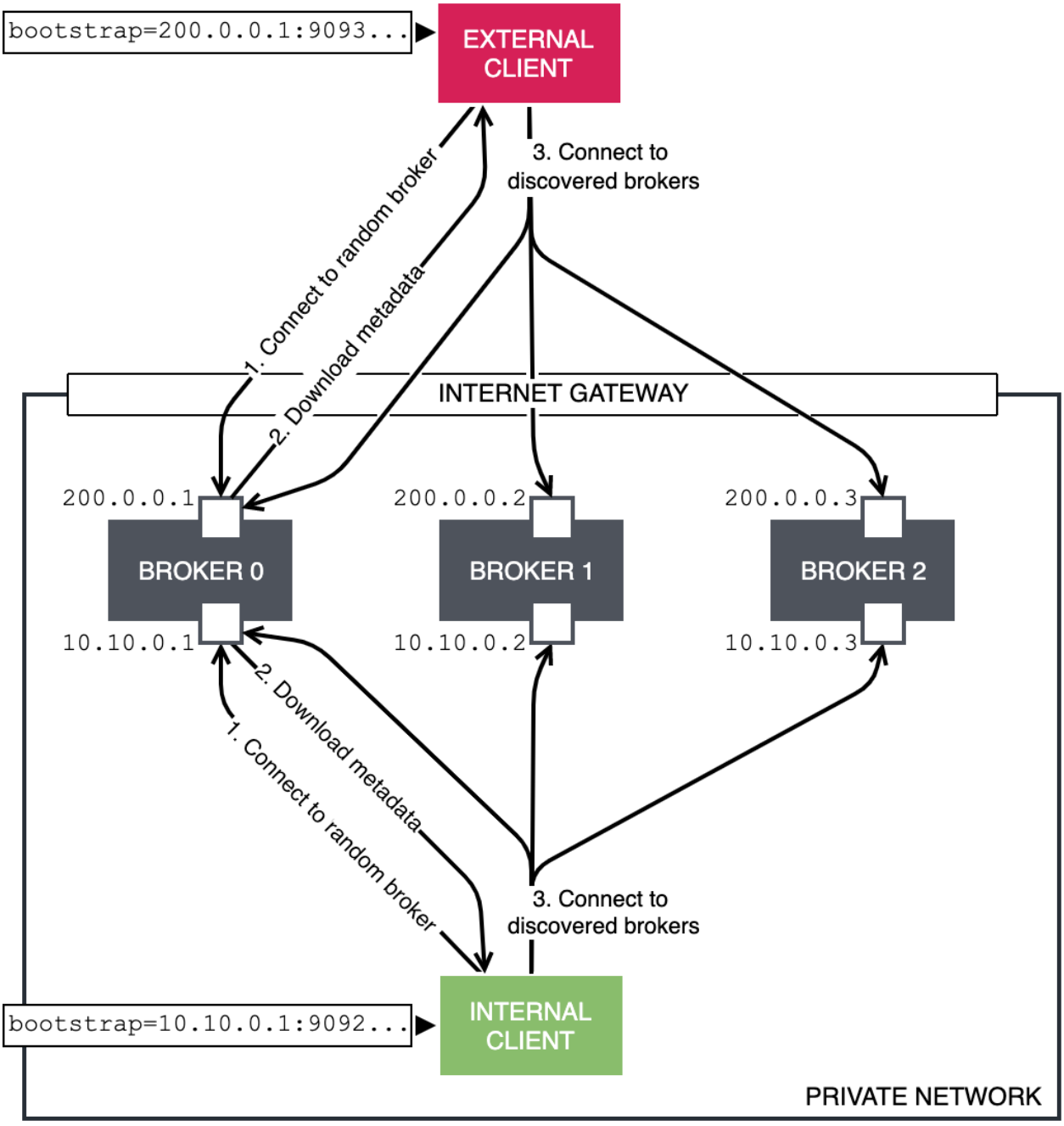
Connecting from two locations

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

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

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

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



Bootstrapping from two locations

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

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

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

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

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

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

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

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

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

Listeners and the Docker Network

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

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

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

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

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

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

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

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

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

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

Now try listing the topics:

[12] http://localhost:9000

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

You should see a single entry echoed to the terminal:

```
test
```

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

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

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

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

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

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

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

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