Apache Kafka® is a distributed streaming platform. What exactly does that mean?
A streaming platform has three key capabilities:

- Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
- Store streams of records in a fault-tolerant durable way.
- Process streams of records as they occur.

Kafka is generally used for two broad classes of applications:

- Building real-time streaming data pipelines that reliably get data between systems or applications
- Building real-time streaming applications that transform or react to the streams of data

To understand how Kafka does these things, let's dive in and explore Kafka's capabilities from the bottom up.

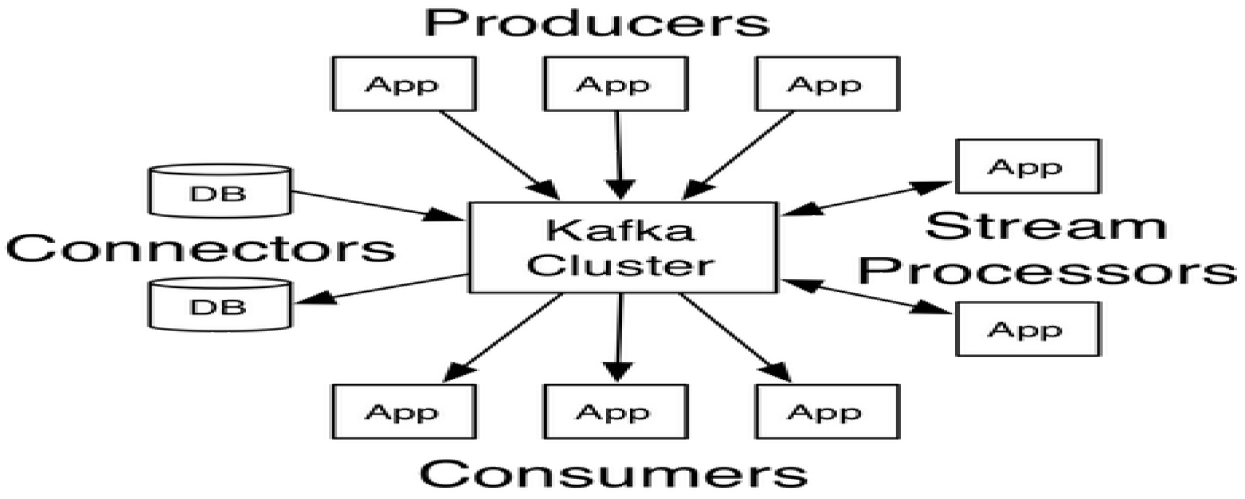
First a few concepts:

- Kafka is run as a cluster on one or more servers that can span multiple datacenters.
- The Kafka cluster stores streams of records in categories called topics.
- Each record consists of a key, a value, and a timestamp.

Kafka has four core APIs:

- The **Producer API** allows an application to publish a stream of records to one or more Kafka topics.
- The **Consumer API** allows an application to subscribe to one or more topics and process the stream of records produced to them.
- The **Streams API** allows an application to act as a stream processor, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.
- The **Connector API** allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table.

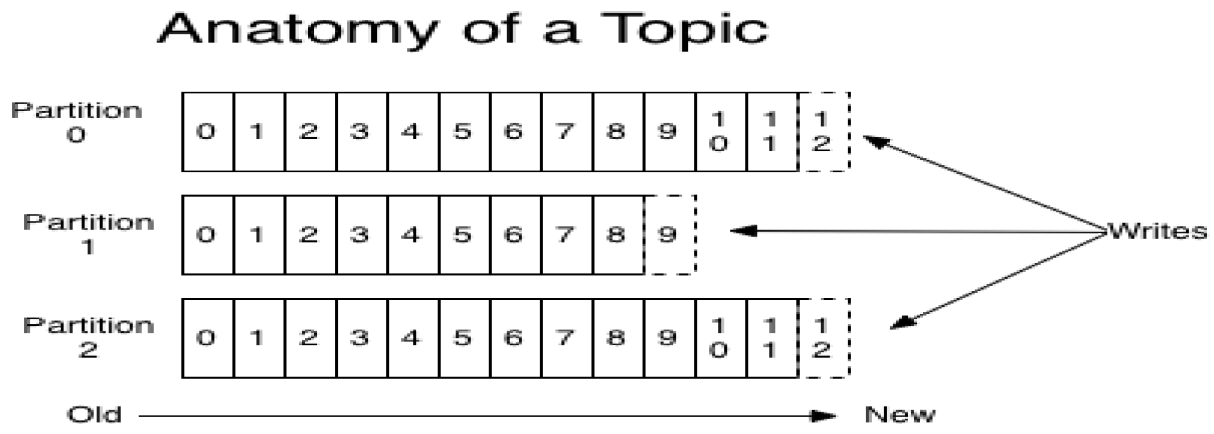
In Kafka the communication between the clients and the servers is done with a simple, high-performance, language agnostic TCP protocol. This protocol is versioned and maintains backwards compatibility with older version. We provide a Java client for Kafka, but clients are available in many languages.



Topics and Logs
Let's first dive into the core abstraction Kafka provides for a stream of records—the **topic**.

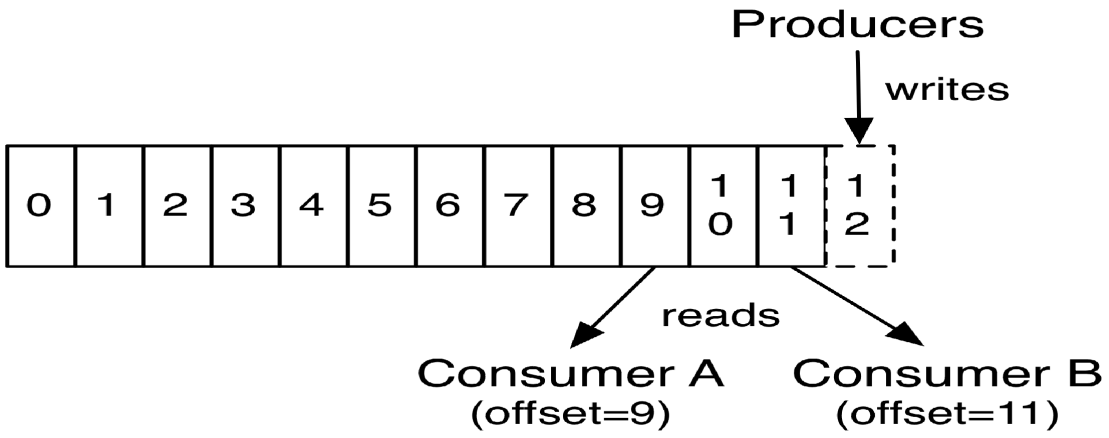
A topic is a category or feed name to which records are published. Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data written to it.

For each topic, the Kafka cluster maintains a partitioned log that looks like this:



Each partition is an ordered, immutable sequence of records that is continually appended to—a structured commit log. The records in the partitions are each assigned a sequential id number called the offset that uniquely identifies each record within the partition.

The Kafka cluster durably persists all published records—whether or not they have been consumed—using a configurable retention period. For example, if the retention policy is set to two days, then for the two days after a record is published, it is available for consumption, after which it will be discarded to free up space. Kafka's performance is effectively constant with respect to data size so storing data for a long time is not a problem.



In fact, the only metadata retained on a per-consumer basis is the offset or position of that consumer in the log. This offset is controlled by the consumer: normally a consumer will advance its offset linearly as it reads records, but, in fact, since the position is controlled by the consumer it can consume records in any order it likes. For example a consumer can reset to an older offset to reprocess data from the past or skip ahead to the most recent record and start consuming from "now".

This combination of features means that Kafka consumers are very cheap—they can come and go without much impact on the cluster or on other consumers. For example, you can use our command line tools to "tail" the contents of any topic without changing what is consumed by any existing consumers.

The partitions in the log serve several purposes. First, they allow the log to scale beyond a size that will fit on a single server. Each individual partition must fit on the servers that host it, but a topic may have many partitions so it can handle an arbitrary amount of data. Second they act as the unit of parallelism—more on that in a bit.

Distribution
The partitions of the log are distributed over the servers in the Kafka cluster with each server handling data and requests for a share of the partitions. Each partition is replicated across a configurable number of servers for fault tolerance.

Each partition has one server which acts as the "leader" and zero or more servers which act as "followers". The leader handles all read and write requests for the partition while the followers passively replicate the leader. If the leader fails, one of the followers will automatically become the new leader. Each server acts as a leader for some of its partitions and a follower for others so load is well balanced within the cluster.

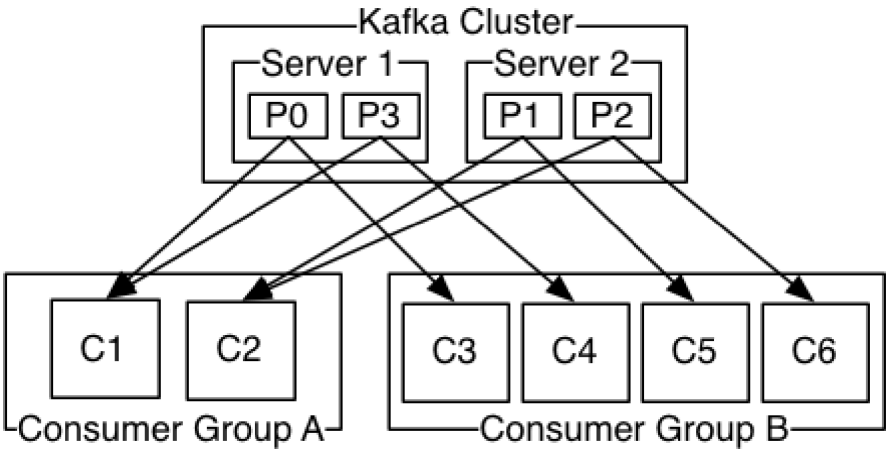
Geo-Replication
Kafka MirrorMaker provides geo-replication support for your clusters. With MirrorMaker, messages are replicated across multiple datacenters or cloud regions. You can use this in active/passive scenarios for backup and recovery; or in active/active scenarios to place data closer to your users, or support data locality requirements.

Producers
Producers publish data to the topics of their choice. The producer is responsible for choosing which record to assign to which partition within the topic. This can be done in a round-robin fashion simply to balance load or it can be done according to some semantic partition function (say based on some key in the record). More on the use of partitioning in a second!

Consumers
Consumers label themselves with a consumer group name, and each record published to a topic is delivered to one consumer instance within each subscribing consumer group. Consumer instances can be in separate processes or on separate machines.

If all the consumer instances have the same consumer group, then the records will effectively be load balanced over the consumer instances.

If all the consumer instances have different consumer groups, then each record will be broadcast to all the consumer processes.



A two server Kafka cluster hosting four partitions (P0-P3) with two consumer groups. Consumer group A has two consumer instances and group B has four.

More commonly, however, we have found that topics have a small number of consumer groups, one for each "logical subscriber". Each group is composed of many consumer instances for scalability and fault tolerance. This is nothing more than publish-subscribe semantics where the subscriber is a cluster of consumers instead of a single process.

The way consumption is implemented in Kafka is by dividing up the partitions in the log over the consumer instances so that each instance is the exclusive consumer of a "fair share" of partitions at any point in time. This process of maintaining membership in the group is handled by the Kafka protocol dynamically. If new instances join the group they will take over some partitions from other members of the group; if an instance dies, its partitions will be distributed to the remaining instances.

Kafka only provides a total order over records within a partition, not between different partitions in a topic. Per-partition ordering combined with the ability to partition data by key is sufficient for most applications. However, if you require a total order over records this can be achieved with a topic that has only one partition, though this will mean only one consumer process per consumer group.

Multi-tenancy
You can deploy Kafka as a multi-tenant solution. Multi-tenancy is enabled by configuring which topics can produce or consume data. There is also operations support for quotas. Administrators can define and enforce quotas on requests to control the broker resources that are used by clients.

Guarantees
At a high-level Kafka gives the following guarantees:

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent. That is, if a record M1 is sent by the same producer as a record M2, and M1 is sent first, then M1 will have a lower offset than M2 and appear earlier in the log.
- A consumer instance sees records in the order they are stored in the log.
- For a topic with replication factor N, we will tolerate up to N-1 server failures without losing any records committed to the log.

Kafka as a Messaging System
How does Kafka's notion of streams compare to a traditional enterprise messaging system?

Messaging traditionally has two models: queuing and publish-subscribe. In a queue, a pool of consumers may read from a server and each record goes to one of them; in publish-subscribe the record is broadcast to all consumers. Each of these two models has a strength and a weakness. The strength of queuing is that it allows you to divide up the processing of data over multiple consumer instances, which lets you scale your processing. Unfortunately, queues aren't multi-subscriber—once one process reads the data it's gone. Publish-subscribe allows you broadcast data to multiple processes, but has no way of scaling processing since every message goes to every subscriber.

The consumer group concept in Kafka generalizes these two concepts. As with a queue the consumer group allows you to divide up processing over a collection of processes (the members of the consumer group). As with publish-subscribe, Kafka allows you to broadcast messages to multiple consumer groups.

The advantage of Kafka's model is that every topic has both these properties—it can scale processing and is also multi-subscriber—there is no need to choose one or the other.

Kafka has stronger ordering guarantees than a traditional messaging system, too.

A traditional queue retains records in-order on the server, and if multiple consumers consume from the queue then the server hands out records in the order they are stored. However, although the server hands out records in order, the records are delivered asynchronously to consumers, so they may arrive out of order on different consumers. This effectively means the ordering of the records is lost in the presence of parallel consumption. Messaging systems often work around this by having a notion of "exclusive consumer" that allows only one process to consume from a queue, but of course this means that there is no parallelism in processing.

Kafka does it better. By having a notion of parallelism—the partition—within the topics, Kafka is able to provide both ordering guarantees and load balancing over a pool of consumer processes. This is achieved by assigning the partitions in the topic to the consumers in the consumer group so that each partition is consumed by exactly one consumer in the group. By doing this we ensure that the consumer is the only reader of that partition and consumes the data in order. Since there are many partitions this still balances the load over many consumer instances. Note however that there cannot be more consumer instances in a consumer group than partitions.

Kafka as a Storage System
Any message queue that allows publishing messages decoupled from consuming them is effectively acting as a storage system for the in-flight messages. What is different about Kafka is that it is a very good storage system.

Data written to Kafka is written to disk and replicated for fault-tolerance. Kafka allows producers to wait on acknowledgement so that a write isn't considered complete until it is fully replicated and guaranteed to persist even if the server written to fails.

The disk structures Kafka uses scale well—Kafka will perform the same whether you have 50 KB or 50 TB of persistent data on the server.

As a result of taking storage seriously and allowing the clients to control their read position, you can think of Kafka as a kind of special purpose distributed filesystem dedicated to high-performance, low-latency commit log storage, replication, and propagation.

Kafka for Stream Processing
It isn't enough to just read, write, and store streams of data, the purpose is to enable real-time processing of streams.

In Kafka a stream processor is anything that takes continual streams of data from input topics, performs some processing on this input, and produces continual streams of data to output topics.

For example, a retail application might take in input streams of sales and shipments, and output a stream of reorders and price adjustments computed off this data.

It is possible to do simple processing directly using the producer and consumer APIs. However for more complex transformations Kafka provides a fully integrated Streams API. This allows building applications that do non-trivial processing that compute aggregations off of streams or join streams together.

This facility helps solve the hard problems this type of application faces: handling out-of-order data, reprocessing input as code changes, performing stateful computations, etc.

The streams API builds on the core primitives Kafka provides: it uses the producer and consumer APIs for input, uses Kafka for stateful storage, and uses the same group mechanism for fault tolerance among the stream processor instances.

Putting the Pieces Together
This combination of messaging, storage, and stream processing may seem unusual but it is essential to Kafka's role as a streaming platform.

A distributed file system like HDFS allows storing static files for batch processing. Effectively a system like this allows storing and processing historical data from the past.

A traditional enterprise messaging system allows processing future messages that will arrive after you subscribe. Applications built in this way process future data as it arrives.

Kafka combines both of these capabilities, and the combination is critical both for Kafka usage as a platform for streaming applications as well as for streaming data pipelines.

By combining storage and low-latency subscriptions, streaming applications can treat both past and future data the same way. That is a single application can process historical, stored data but rather than ending when it reaches the last record it can keep processing as future data arrives. This is a generalized notion of stream processing that subsumes batch processing as well as message-driven applications.

Likewise for streaming data pipelines the combination of subscription to real-time events make it possible to use Kafka for very low-latency pipelines; but the ability to store data reliably make it possible to use it for critical data where the delivery of data must be guaranteed or for integration with offline systems that load data only periodically or may go down for extended periods of time for maintenance.

The stream processing facilities make it possible to transform data as it arrives.

1.2 Use Cases

Here is a description of a few of the popular use cases for Apache Kafka®.

Messaging

Kafka works well as a replacement for a more traditional message broker. Message brokers are used for a variety of reasons (to decouple processing from data producers, to buffer unprocessed messages, etc). In comparison to most messaging systems Kafka has better throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large scale message processing applications.

In our experience messaging uses are often comparatively low-throughput, but may require low end-to-end latency and often depend on the strong durability guarantees Kafka provides.

In this domain Kafka is comparable to traditional messaging systems such as ActiveMQ or RabbitMQ.

Website Activity Tracking

The original use case for Kafka was to be able to rebuild a user activity tracking pipeline as a set of real-time publish-subscribe feeds. This means site activity (page views, searches, or other actions users may take) is published to central topics with one topic per activity type. These feeds are available for subscription for a range of use cases including real-time processing, real-time monitoring, and loading into Hadoop or offline data warehousing systems for offline processing and reporting.

Activity tracking is often very high volume as many activity messages are generated for each user page view.

Metrics

Kafka is often used for operational monitoring data. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.

Log Aggregation

Many people use Kafka as a replacement for a log aggregation solution. Log aggregation typically collects physical log files off servers and puts them in a central place (a file server or HDFS perhaps) for processing. Kafka abstracts away the details of files and gives a cleaner abstraction of log or event data as a stream of messages. This allows for lower-latency processing and easier support for multiple data sources and distributed data consumption. In comparison to log-centric systems like Scribe or Flume, Kafka offers equally good performance, stronger durability guarantees due to replication, and much lower end-to-end latency.

Stream Processing

Many users of Kafka process data in processing pipelines consisting of multiple stages, where raw input data is consumed from Kafka topics and then aggregated, enriched, or otherwise transformed into new topics for further consumption or follow-up processing. For example, a processing pipeline for recommending news articles might crawl article content from RSS feeds and publish it to an "articles" topic; further processing might normalize or deduplicate this content and published the cleansed article content to a new topic; a final processing stage might attempt to recommend this content to users. Such processing pipelines create graphs of real-time data flows based on the individual topics. Starting in 0.10.0.0, a light-weight but powerful stream processing library called Kafka Streams is available in

Apache Kafka to perform such data processing as described above. Apart from Kafka Streams, alternative open source stream processing tools include Apache Storm and Apache Samza.

Event Sourcing
Event sourcing is a style of application design where state changes are logged as a time-ordered sequence of records. Kafka's support for very large stored log data makes it an excellent backend for an application built in this style.

Commit Log
Kafka can serve as a kind of external commit-log for a distributed system. The log helps replicate data between nodes and acts as a re-syncing mechanism for failed nodes to restore their data. The log compaction feature in Kafka helps support this usage. In this usage Kafka is similar to Apache BookKeeper project.

1.3 Quick Start
This tutorial assumes you are starting fresh and have no existing Kafka or ZooKeeper data. Since Kafka console scripts are different for Unix-based and Windows platforms, on Windows platforms use bin\windows\ instead of bin/, and change the script extension to .bat.

Step 1: Download the code
Download the 2.2.0 release and un-tar it.

```
> tar -xzf kafka_2.12-2.2.0.tgz
> cd kafka_2.12-2.2.0
```

Step 2: Start the server
Kafka uses ZooKeeper so you need to first start a ZooKeeper server if you don't already have one. You can use the convenience script packaged with kafka to get a quick-and-dirty single-node ZooKeeper instance.

```
> bin/zookeeper-server-start.sh config/zookeeper.properties
[2013-04-22 15:01:37,495] INFO Reading configuration from: config/zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
...
```

Now start the Kafka server:
```
> bin/kafka-server-start.sh config/server.properties
[2013-04-22 15:01:47,028] INFO Verifying properties (kafka.utils.VerifiableProperties)
[2013-04-22 15:01:47,051] INFO Property socket.send.buffer.bytes is overridden to 1048576 (kafka.utils.VerifiableProperties)
...
```

Step 3: Create a topic
Let's create a topic named "test" with a single partition and only one replica:

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

We can now see that topic if we run the list topic command:

```
> bin/kafka-topics.sh --list --bootstrap-server localhost:9092

test
```
Alternatively, instead of manually creating topics you can also configure your brokers to auto-create topics when a non-existent topic is published to.

Step 4: Send some messages
Kafka comes with a command line client that will take input from a file or from standard input and send it out as messages to the Kafka cluster. By default, each line will be sent as a separate message.
Run the producer and then type a few messages into the console to send to the server.

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
This is a message
This is another message
```

Step 5: Start a consumer
Kafka also has a command line consumer that will dump out messages to standard output.
```
> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning
This is a message
This is another message
```

If you have each of the above commands running in a different terminal then you should now be able to type messages into the producer terminal and see them appear in the consumer terminal.

All of the command line tools have additional options; running the command with no arguments will display usage information documenting them in more detail.

Step 6: Setting up a multi-broker cluster
So far we have been running against a single broker, but that's no fun. For Kafka, a single broker is just a cluster of size one, so nothing much changes other than starting a few more broker instances. But just to get feel for it, let's expand our cluster to three nodes (still all on our local machine).

First we make a config file for each of the brokers (on Windows use the copy command instead):
```
> cp config/server.properties config/server-1.properties
> cp config/server.properties config/server-2.properties
```

Now edit these new files and set the following properties:
config/server-1.properties:
```
   broker.id=1
   listeners=PLAINTEXT://:9093
   log.dirs=/tmp/kafka-logs-1
```

config/server-2.properties:
```
   broker.id=2
   listeners=PLAINTEXT://:9094
   log.dirs=/tmp/kafka-logs-2
```

The broker.id property is the unique and permanent name of each node in the cluster. We have to override the port and log directory only because we are running these all on the same machine and we want

to keep the brokers from all trying to register on the same port or overwrite each other's data.

We already have Zookeeper and our single node started, so we just need to start the two new nodes:
```
> bin/kafka-server-start.sh config/server-1.properties &
...
> bin/kafka-server-start.sh config/server-2.properties &
...
```

Now create a new topic with a replication factor of three:
```
> bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 3 --partitions 1 --topic my-replicated-topic
```

Okay but now that we have a cluster how can we know which broker is doing what? To see that run the "describe topics" command:
```
> bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 --topic my-replicated-topic
Topic:my-replicated-topic   PartitionCount:1    ReplicationFactor:3 Configs:
    Topic: my-replicated-topic  Partition: 0    Leader: 1   Replicas: 1,2,0 Isr: 1,2,0
```

Here is an explanation of output. The first line gives a summary of all the partitions, each additional line gives information about one partition. Since we have only one partition for this topic there is only one line.

"leader" is the node responsible for all reads and writes for the given partition. Each node will be the leader for a randomly selected portion of the partitions.
"replicas" is the list of nodes that replicate the log for this partition regardless of whether they are the leader or even if they are currently alive.
"isr" is the set of "in-sync" replicas. This is the subset of the replicas list that is currently alive and caught-up to the leader.

Note that in my example node 1 is the leader for the only partition of the topic.

We can run the same command on the original topic we created to see where it is:

```
> bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 --topic test
Topic:test  PartitionCount:1    ReplicationFactor:1 Configs:
    Topic: test Partition: 0    Leader: 0   Replicas: 0 Isr: 0
```

So there is no surprise there—the original topic has no replicas and is on server 0, the only server in our cluster when we created it.

Let's publish a few messages to our new topic:
```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic my-replicated-topic
...
my test message 1
my test message 2
^C
```

Now let's consume these messages:
```
> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --from-beginning --topic my-repli-
```

cated-topic
...
my test message 1
my test message 2
^C

Now let's test out fault-tolerance. Broker 1 was acting as the leader so let's kill it:

> ps aux | grep server-1.properties
7564 ttys002 0:15.91 /System/Library/Frameworks/JavaVM.framework/Versions/1.8/Home/bin/java...
> kill -9 7564

On Windows use:
> wmic process where "caption = 'java.exe' and commandline like '%server-1.properties%'" get processid
ProcessId
6016
> taskkill /pid 6016 /f

Leadership has switched to one of the followers and node 1 is no longer in the in-sync replica set:
> bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 --topic my-replicated-topic
Topic:my-replicated-topic PartitionCount:1 ReplicationFactor:3 Configs:
Topic: my-replicated-topic Partition: 0 Leader: 2 Replicas: 1,2,0 Isr: 2,0

But the messages are still available for consumption even though the leader that took the writes originally is down:
> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --from-beginning --topic my-replicated-topic
...
my test message 1
my test message 2
^C

Step 7: Use Kafka Connect to import/export data
Writing data from the console and writing it back to the console is a convenient place to start, but you'll probably want to use data from other sources or export data from Kafka to other systems. For many systems, instead of writing custom integration code you can use Kafka Connect to import or export data.

Kafka Connect is a tool included with Kafka that imports and exports data to Kafka. It is an extensible tool that runs connectors, which implement the custom logic for interacting with an external system. In this quickstart we'll see how to run Kafka Connect with simple connectors that import data from a file to a Kafka topic and export data from a Kafka topic to a file.

First, we'll start by creating some seed data to test with:

> echo -e "foo\nbar" > test.txt

Or on Windows:

```
> echo foo> test.txt
> echo bar>> test.txt
```

Next, we'll start two connectors running in standalone mode, which means they run in a single, local, dedicated process. We provide three configuration files as parameters. The first is always the configuration for the Kafka Connect process, containing common configuration such as the Kafka brokers to connect to and the serialization format for data. The remaining configuration files each specify a connector to create. These files include a unique connector name, the connector class to instantiate, and any other configuration required by the connector.

```
> bin/connect-standalone.sh config/connect-standalone.properties config/connect-file-source.properties config/connect-file-sink.properties
```

These sample configuration files, included with Kafka, use the default local cluster configuration you started earlier and create two connectors: the first is a source connector that reads lines from an input file and produces each to a Kafka topic and the second is a sink connector that reads messages from a Kafka topic and produces each as a line in an output file.

During startup you'll see a number of log messages, including some indicating that the connectors are being instantiated. Once the Kafka Connect process has started, the source connector should start reading lines from test.txt and producing them to the topic connect-test, and the sink connector should start reading messages from the topic connect-test and write them to the file test.sink.txt. We can verify the data has been delivered through the entire pipeline by examining the contents of the output file:

```
> more test.sink.txt
foo
bar
```

Note that the data is being stored in the Kafka topic connect-test, so we can also run a console consumer to see the data in the topic (or use custom consumer code to process it):

```
> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic connect-test --from-beginning
{"schema":{"type":"string","optional":false},"payload":"foo"}
{"schema":{"type":"string","optional":false},"payload":"bar"}
...
```

The connectors continue to process data, so we can add data to the file and see it move through the pipeline:

```
> echo Another line>> test.txt
```

You should see the line appear in the console consumer output and in the sink file.

Step 8: Use Kafka Streams to process data

Kafka Streams is a client library for building mission-critical real-time applications and microservices, where the input and/or output data is stored in Kafka clusters. Kafka Streams combines the simplicity of writing and deploying standard Java and Scala applications on the client side with the benefits of Kafka's server-side cluster technology to make these applications highly scalable, elastic, fault-tolerant, distributed, and much more. This quickstart example will demonstrate how to run a streaming application coded in this library.

2. APIs

Kafka includes five core apis:

- The Producer API allows applications to send streams of data to topics in the Kafka cluster.
- The Consumer API allows applications to read streams of data from topics in the Kafka cluster.
- The Streams API allows transforming streams of data from input topics to output topics.
- The Connect API allows implementing connectors that continually pull from some source system or application into Kafka or push from Kafka into some sink system or application.
- The AdminClient API allows managing and inspecting topics, brokers, and other Kafka objects.

Kafka exposes all its functionality over a language independent protocol which has clients available in many programming languages. However only the Java clients are maintained as part of the main Kafka project, the others are available as independent open source projects. A list of non-Java clients is available here.

2.1 Producer API

The Producer API allows applications to send streams of data to topics in the Kafka cluster.

Examples showing how to use the producer are given in the javadocs.

To use the producer, you can use the following maven dependency:

```
<dependency>
   <groupId>org.apache.kafka</groupId>
   <artifactId>kafka-clients</artifactId>
   <version>2.2.0</version>
</dependency>
```

2.2 Consumer API

The Consumer API allows applications to read streams of data from topics in the Kafka cluster.

Examples showing how to use the consumer are given in the javadocs.

To use the consumer, you can use the following maven dependency:

```
<dependency>
   <groupId>org.apache.kafka</groupId>
   <artifactId>kafka-clients</artifactId>
   <version>2.2.0</version>
</dependency>
```

2.3 Streams API

The Streams API allows transforming streams of data from input topics to output topics.

Examples showing how to use this library are given in the javadocs

Additional documentation on using the Streams API is available here.

To use Kafka Streams you can use the following maven dependency:

```
<dependency>
   <groupId>org.apache.kafka</groupId>
   <artifactId>kafka-streams</artifactId>
   <version>2.2.0</version>
</dependency>
```

When using Scala you may optionally include the kafka-streams-scala library. Additional documentation on using the Kafka Streams DSL for Scala is available in the developer guide.

To use Kafka Streams DSL for Scala for Scala 2.12 you can use the following maven dependency:

```
<dependency>
   <groupId>org.apache.kafka</groupId>
   <artifactId>kafka-streams-scala_2.12</artifactId>
   <version>2.2.0</version>
</dependency>
```

2.4 Connect API

The Connect API allows implementing connectors that continually pull from some source data system into Kafka or push from Kafka into some sink data system.

Many users of Connect won't need to use this API directly, though, they can use pre-built connectors without needing to write any code. Additional information on using Connect is available here.

Those who want to implement custom connectors can see the javadoc.

2.5 AdminClient API

The AdminClient API supports managing and inspecting topics, brokers, acls, and other Kafka objects.

To use the AdminClient API, add the following Maven dependency:

```
<dependency>
   <groupId>org.apache.kafka</groupId>
   <artifactId>kafka-clients</artifactId>
   <version>2.2.0</version>
</dependency>
```

For more information about the AdminClient APIs, see the javadoc.