Kafka Fundamentals

Ballista
TECHNOLOGY GROUP

Your Instructors: Petter Graff

- Serial entrepreneur, architect, consultant, teacher, and sometime CXX
- Owner/Partner of Ballista Technology Group and CTO of Praetexo,
- Architect of Yaktor, a scalable event-driven, agent based rapid development platform
- Teaches classes on:
  - BigData (Hadoop, Spark, HBase, Cassandra, etc.)
  - Computer Science Concepts (Scalability, Architectural Thinking, Design Patterns, OOA&D, etc.)
  - Languages (Java, Scala, C++, JavaScript, ...)
  - And much more...
- Lives in Austin
- O'Reilly author:
  Design Patterns in Java

Ballista Technology Group

- Accelerator for scale-ups, including:
  - Definition of architecture
  - Custom software development
  - Strategic advice to the C-suite
- Consultant, and mentor to many large organizations worldwide
- Big Data and Machine Learning
- Custom training to leading firms worldwide

- http://www.ballista.com
- Solving the Hardest Problems

PRATEXO

- Helping companies move algorithms to the edge
  - Orchestration of edge solution
  - Setup of private clouds
    - On premise
    - On edge computing nodes
  - Swarm computing for ad-hoc collaboration of edge nodes
- Accelerates the move from months to days using a set of cloud deployed tools

2021

Outline

- Introduction to Kafka
- Kafka core concepts
- Producing data to Kafka using the Producer API
- Consuming data from Kafka using the Consumer API
- Kafka Streaming
- Kafka Administration and Integration
- Exactly once delivery, what does that mean?

What is Kafka?

- *"Kafka is a distributed, partitioned, replicated commit log service"*
  - Fault tolerant
  - Near linearly scalable (horizontally)
  - Durable
- Often used as a publish-subscribe messaging system
- Apache project
- Originally developed by LinkedIn

Ballista

Kafka History

- Publish/subscribe system with an interface where
  - Publishing is similar to a typical messaging system
  - Subscription divert as it works on batches
  - A storage layer like a log aggregation system

- As of August 2015, LinkedIn produces over one trillion messages and a petabyte of data consumed daily!

- Kafka was released as open source in late 2010 and became an Apache project in 2011

The Name

I thought that since Kafka was a system optimized for writing using a writer's name would make sense. I had taken a lot of lit classes in college and liked Franz Kafka. Plus the name sounded cool for an open source project.

So basically there is not much of a relationship.

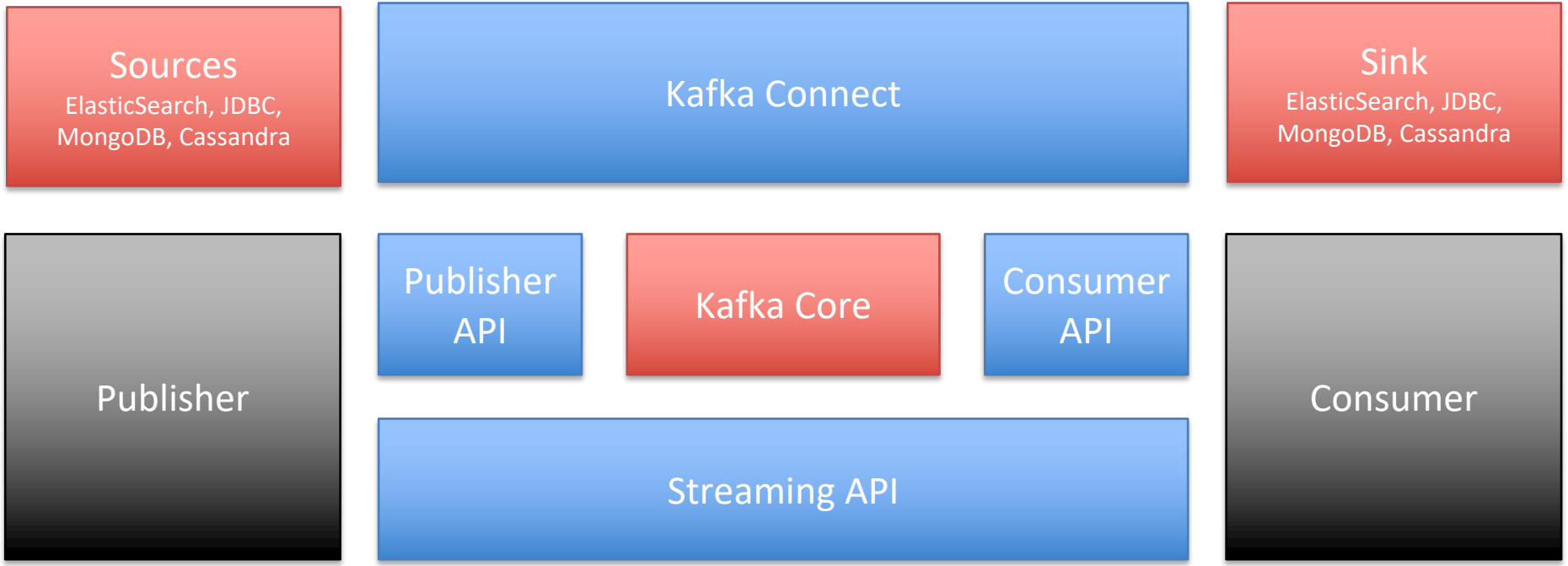
Jay Kreps, Lead Developer at LinkedIn

Notable Users

- Cisco Systems
- Netflix
- PayPal
- Spotify
- Uber
- HubSpot
- Betfair
- Shopify

Ballista

Kafka API's



Ballista

Use Cases

- Messaging
- Website Activity Tracking
- Metrics
- Log Aggregation
- Stream Processing
- Event Sourcing
- Commit Log

Website Activity Tracking

- The original use case for Kafka was to be able to rebuild user activity as a set of real-time publish-subscribe feeds
- Site activity such as page views, searches, or other user actions are published to central topics with one topic per activity type
- These feeds are then available for subscription for processing, monitoring, and loading into offline processing/reporting
- Activity tracking can be **very high volume** as many messages are generated for each user page view

Ballista

Messaging

- Kafka can be a replacement for a more traditional message broker such as ActiveMQ or RabbitMQ
  - But only for publish-subscribe type use cases

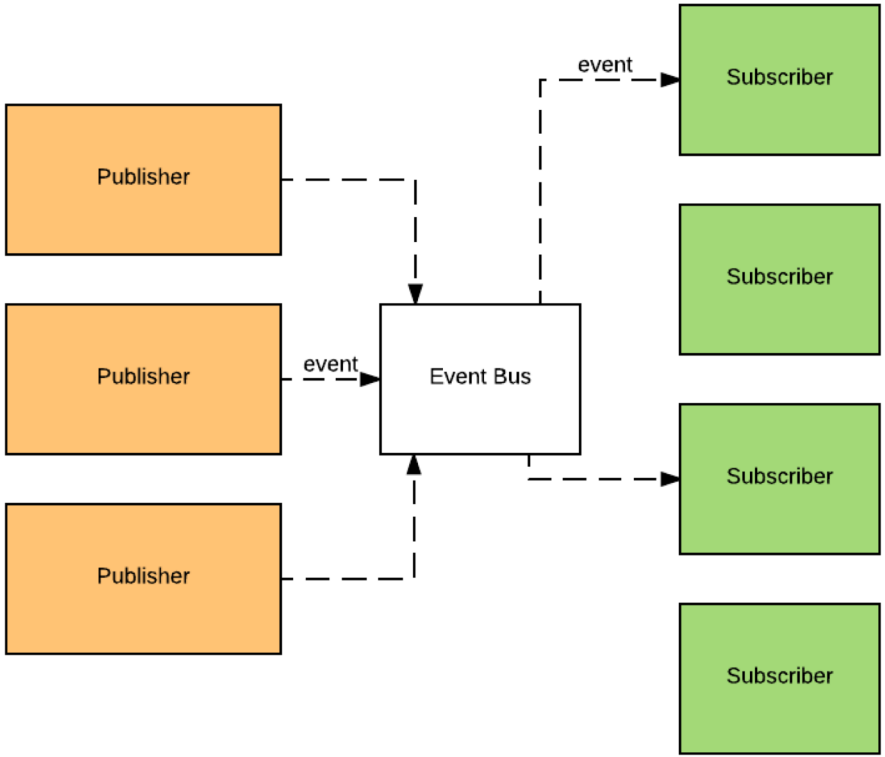
- Message broker scan be used to decouple processing from data producers, buffer unprocessed messages, etc.

- Kafka has replication, built-in partitioning, and fault-tolerance making it a good solution for large scale applications

Publish-Subscribe Pattern

- Sender (publisher) wants to send a piece of data (message)
- The message is not specifically directed to a receiver (subscriber)
- The sender classifies the message and the receiver subscribes to receive certain classes of messages
- Usually there is a central broker where messages are published

Publish-Subscribe Illustrated



Ballista

Log Aggregation

- Log aggregation collects physical log files off servers and put them in a central place such as a file server of HDFS for processing

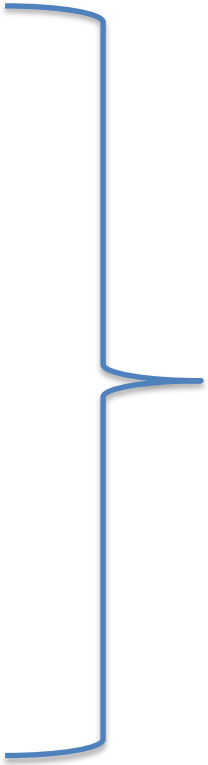
- Kafka abstracts away the details of files and gives a cleaner abstraction of log or event data as a stream of messages

Why Kafka?

- Able to connect a large number of clients (Producers and Consumers)
- Durable
  - Disk based retention
  - Data is replicated across brokers
- Scalable
  - Expansions can be performed while the cluster is online
- High Performance
  - Producers, consumers, and brokers can all be scaled to handle very large message streams
  - Sub-second message latency to consumers

How Big is Big?

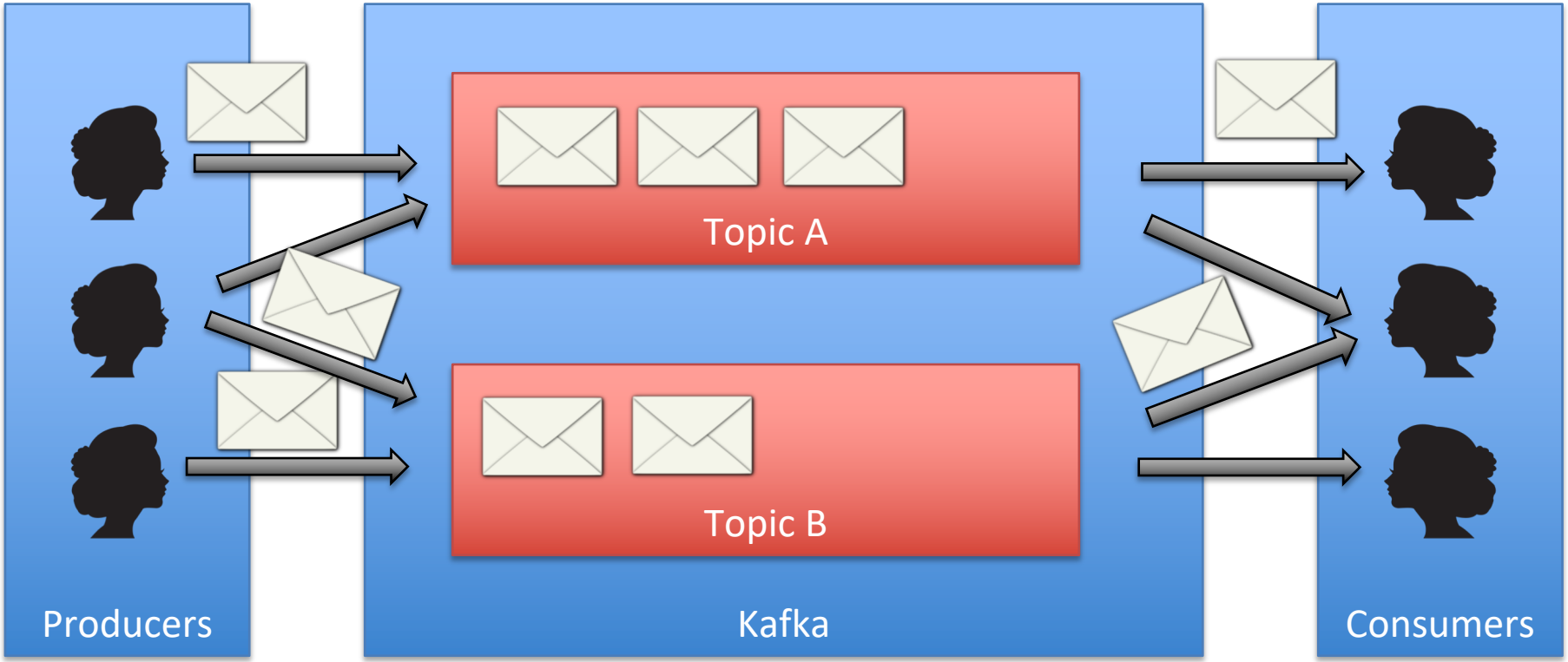
- Per day:
  - 800 billion messages
  - 175 TB of data
  - 650 TB of consumed data
- Per second:
  - 13 million messages
  - 2.75 GB of data
- Configuration
  - 1100 Kafka brokers
  - 60 clusters



Numbers from LinkedIn

Ballista

Kafka Main Concepts: Topics



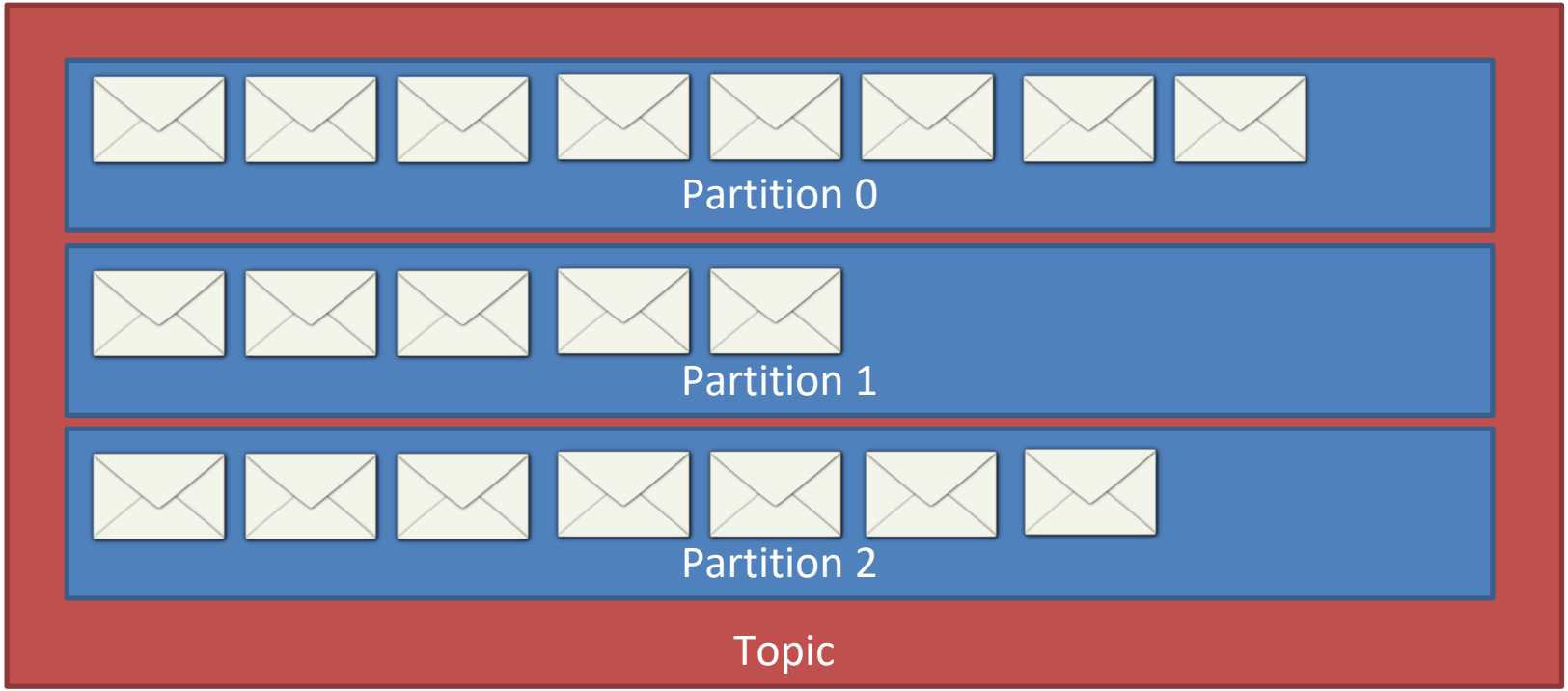
Ballista

Kafka Main Concepts: Message

- Kafka looks at every message as a sequence of bytes
- The bytes are separated into:
  - Key
    - Used to determine partition (more about that later)
  - Value
    - The actual payload of the message

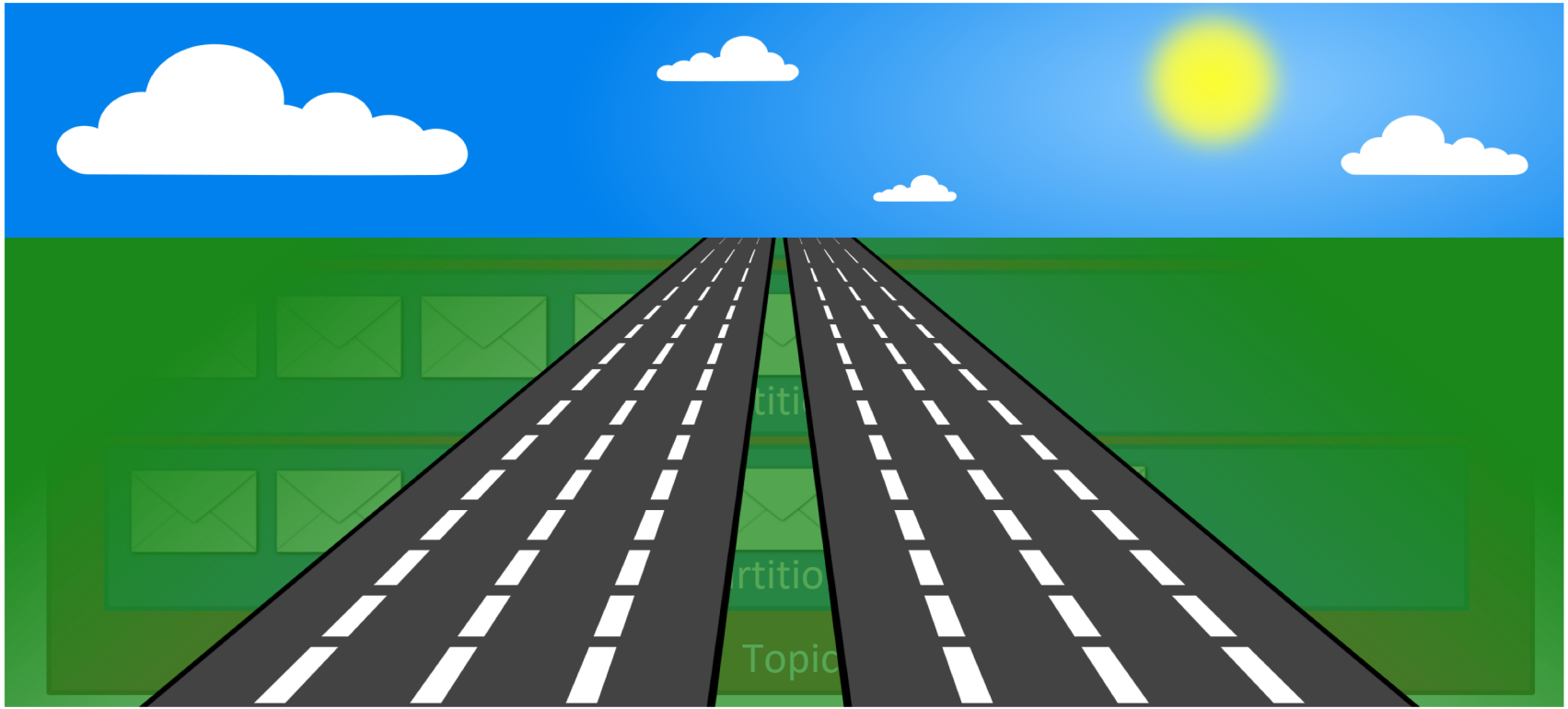
Ballista

Kafka Key Concepts: Partitions



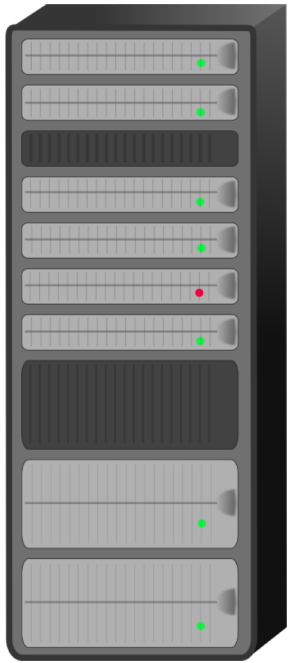
Ballista

Kafka Key Concepts: Partitions



Ballista

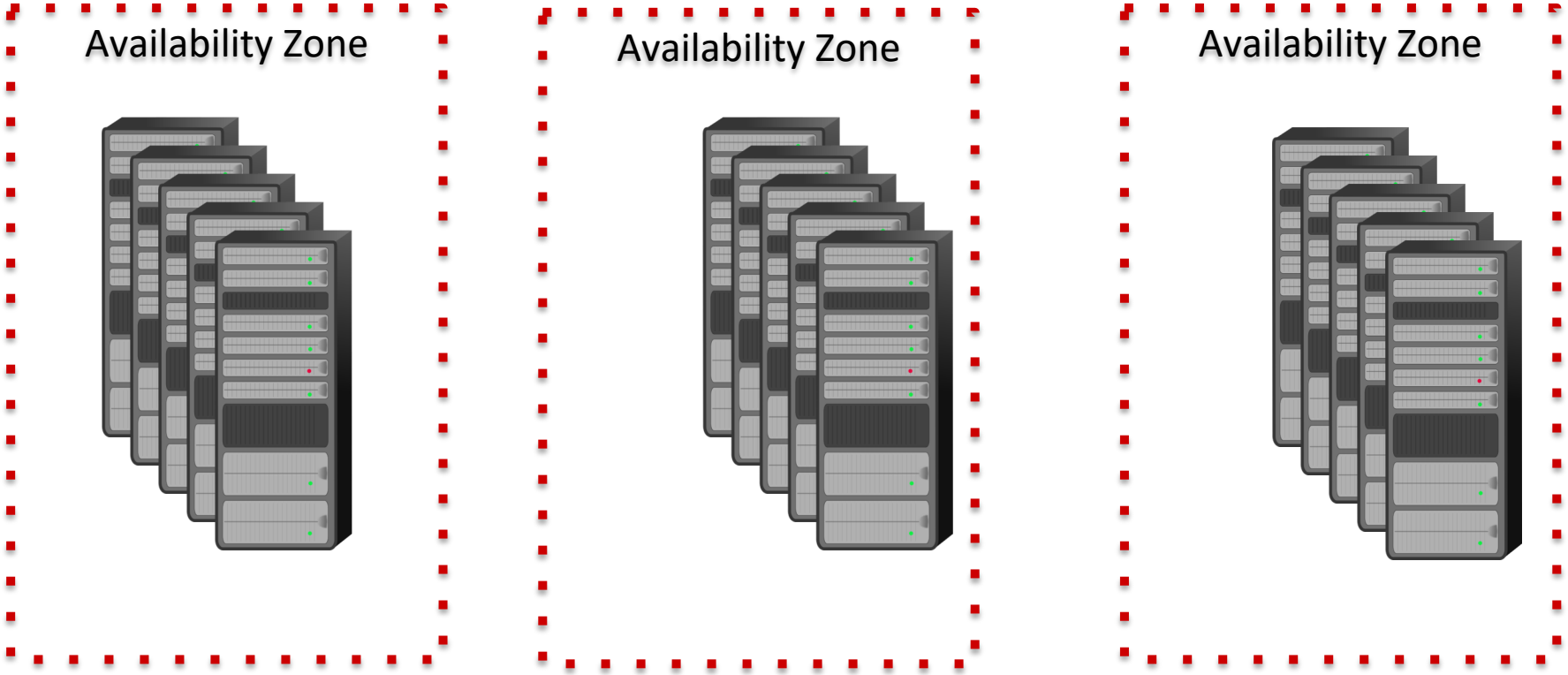
Kafka Main Concepts: Broker



- A server or process running Kafka
- Holds multiple partitions across multiple topics
- The physical manifestation of Kafka

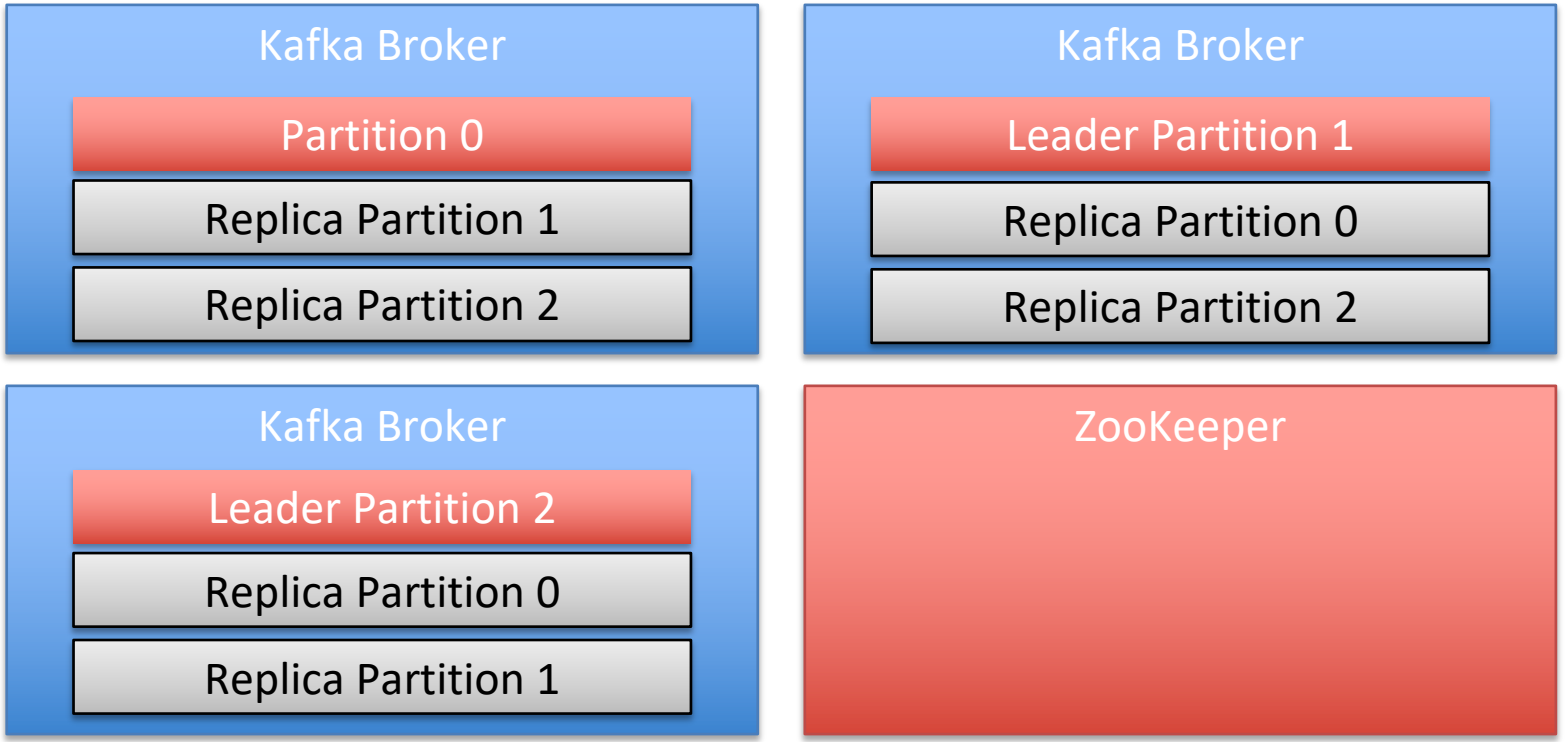
Ballista

Kafka Main Concepts: Cluster



Ballista

Kafka Main Concepts: Partitions and Clusters



Ballista

Summary

- Apache Kafka is an open source, distributed, partitioned, and replicated commit-log based publish-subscribe messaging system
  - Scalable
  - High Performance
  - Multiple Consumers
  - Multiple Producers
  - Disk-based Retention

Ballista

Start of Exercise

- Do you have Docker installed?

- We will be using Docker to run Kafka
- If you have not done so already, please install Docker
  - https://docs.docker.com/engine/installation/

- Check if Docker works by running

Check installation of Docker

```
$ docker -v
Docker version 1.13.0, build 49bf474
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
78445dd45222: Already exists
Digest: sha256:c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://cloud.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

Ballista

docker-compose.yml

```
version: '2'
services:
        zookeeper:
                image: zookeeper:XXX
        kafka:
                image: wurstmeister/kafka:XXX
                environment:
                        HOSTNAME_COMMAND: "echo
$HOSTNAME"
                        KAFKA_ADVERTISED_PORT: 9092
                        KAFKA_ZOOKEEPER_CONNECT:
zookeeper:2181
```

Run Zookeeper

Run Kafka

Ballista

Essential Kafka CLI Commands (used in exercise)

- Kafka can be configured via the command
- Key commands
  - kafka-topics.sh
    - Allows us to manipulate and view topics
  - kafka-console-producer.sh
    - Allows us to produce data from using stdin
  - kafka-console-consumer.sh
    - Allows us to consume messages from the console
- We're running all tools in docker, so we have to 'reach into' the docker instance, hence our commands look like this:
  - docker-compose exec kafka /opt/kafka/bin/CMD
  - You may want to alias these commands to reduce typing or simply run a bash shell inside the docker image
    - docker-compose exec kafka /bin/bash

Ballista

Lab

- In this lab we'll simply ensure that we can run Kafka
- We'll run Kafka using Docker
  - Easy configuration
  - Runs Kafka and Zookeeper
  - Flexible setup that allows us to setup a cluster of machines for later exercises
- The lab simply:
  1. Starts the docker images using docker-compose
  2. Runs a simple producer
  3. Runs a simple consumer
  4. Allows you to type messages in the producer and see them consumed by the consumer

Lab/Demo

The lab description can be found on GitHub in the directory **labs/01-Verify-Installation/hello-world-kafka.md**

Goals of the lab:

- **Make sure your docker environment is working properly**
- **Make sure you can run the Kafka installation in Docker**
- **Show some of the tools to create topics, publish records, and consume records**

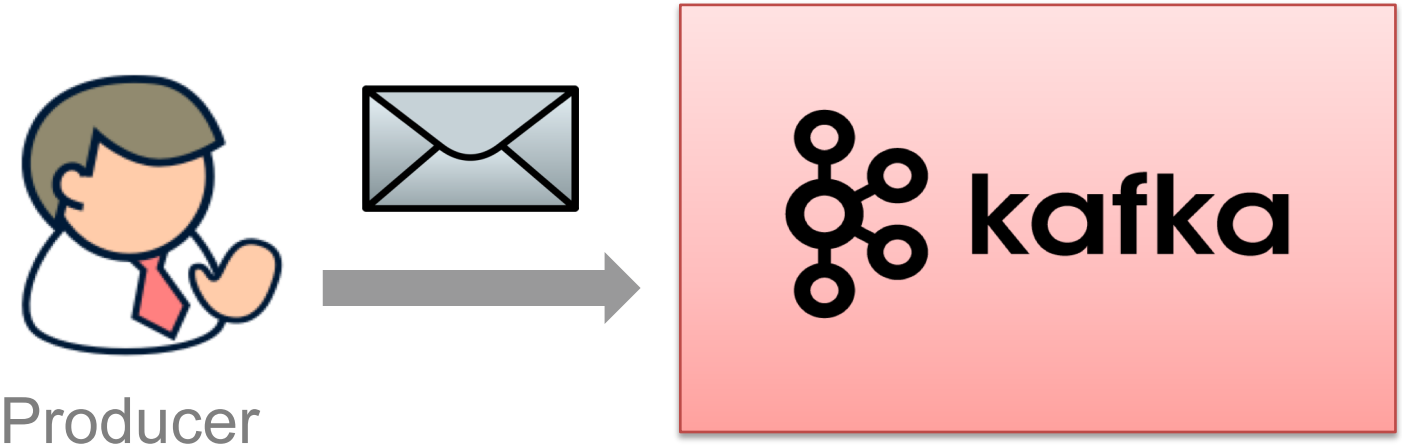
Producers and Consumers

Ballista
TECHNOLOGY GROUP

Outline

- Producers
  - What is a producer?
  - The producer API
- What is a Consumer?
  - What is a consumer and a consumer group?
  - The consumer API
- Anatomy of messages

What is a Producer?

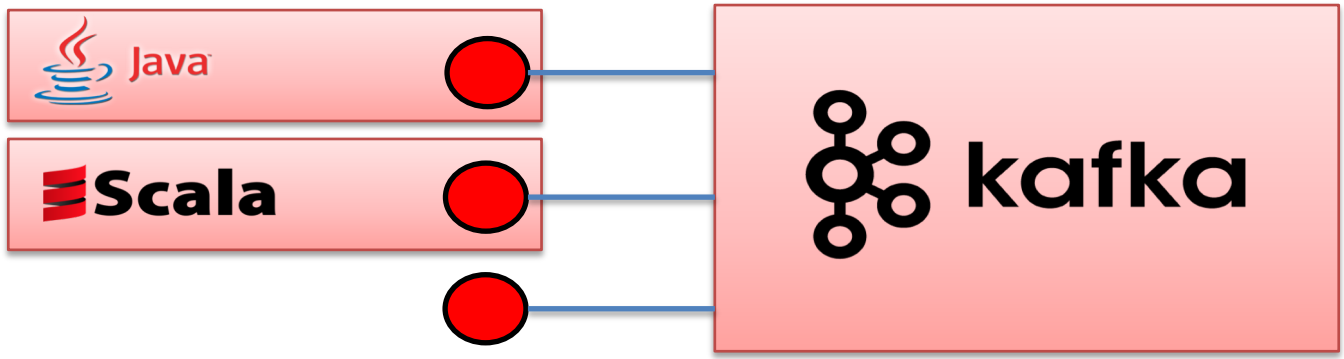


- Producer

- Producers produces the data sent to the Kafka clusters
  - Sent via topics
  - Directly involved in load-balancing
  - Controls the resiliency of messages

Ballista

Kafka APIs



- Kafka ships with built in client APIs for developers to use with applications
  - Kafka ships with a Java client that is recommended
  - Legacy Scala clients are still included
  - Kafka also includes a **binary wire protocol**
  - Many tools in other languages that implement this wire protocol

Ballista

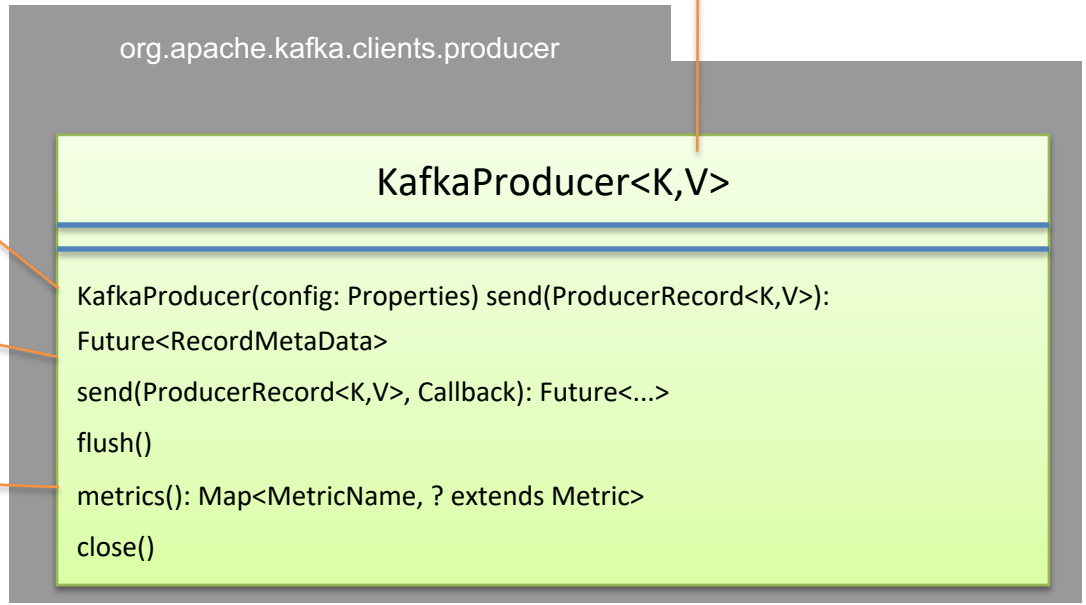
The Java API

Generic sender where:
K = Type of key
V = Type of message

Constructor takes a configuration (mostly a hashmap of options)

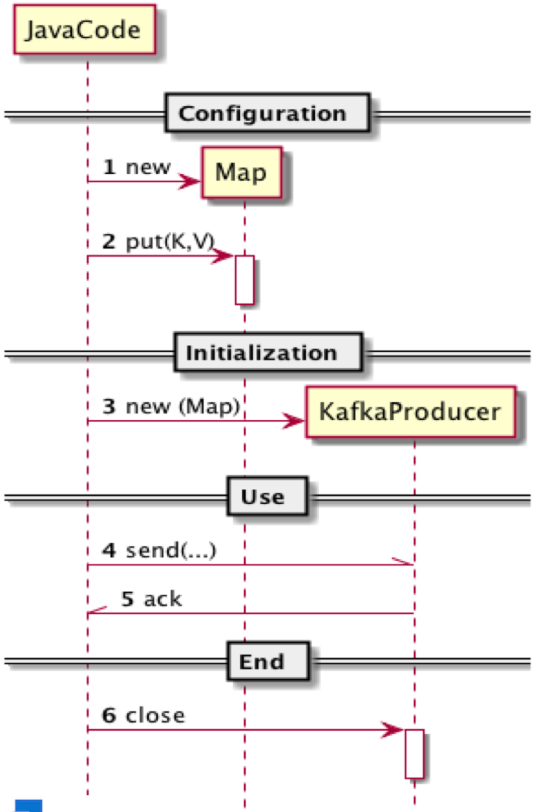
Send a messages (with or without callbacks)

Get metrics for this producer



Ballista

Java API Behavior




```
// Configuration
Properties kp = new Properties();
kp.put("bootstrap.servers",
                "mybroker1:9092,mybroker2:9092");
kp.put("key.serializer", "...");



// Initialization
KafkaProducer<String, String> producer =
                new KafkaProducer<String, String>(kp);

// Use
Future<...> f = producer.send(...);
... f.get(); // when acked


// End
producer.close();
```

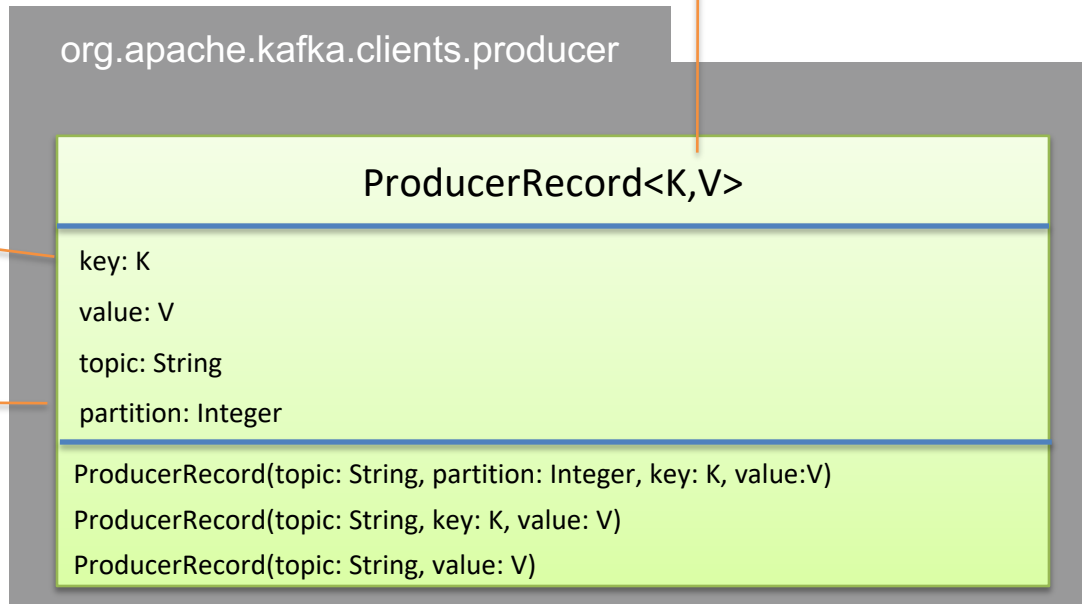
Ballista

Creating a Kafka Producer

- Constructing a Kafka producer requires 3 mandatory properties
  - bootstrap.servers – list of host:port pairs of Kafka brokers. This doesn't have to include all brokers in the cluster as the producer will query about additional brokers. It is recommended to include at least 2 in case one broker goes down
  - key.serializer – should be set to a class that implements the Serializer interface that will be used to serialize **keys**
  - value.serializer - should be set to a class that implements the Serializer interface that will be used to serialize **values**

The ProducerRecord

Generic record where:
K = message key
V = Type of message



Message Key is optional

Partition Key is optional

Ballista

Sending a Message

```
ProducerRecord<String,String> record =
  new ProducerRecord<String,String>(
    "someTopic", "someKey, "someValue");

producer.send(record,  new Callback() {
   public void onCompletion(
      RecordMetadata metadata, Exception e) {
         if(e != null) e.printStackTrace();
         System.out.println(
           "Offset: " + metadata.offset());
      }
});
```

Ballista

Producer Controls Message Guarantees

- As a producer you can determine what guarantees you want Kafka to give you when sending a message
  - Controlled by acknowledgement
- Different use cases requires different guarantees
  - Web page clicks log:
    - Don't care if I lose a few messages
  - Credit card payment
    - I want best possible guarantee
- Kafka provides options
  - The better guarantee, the higher the latency

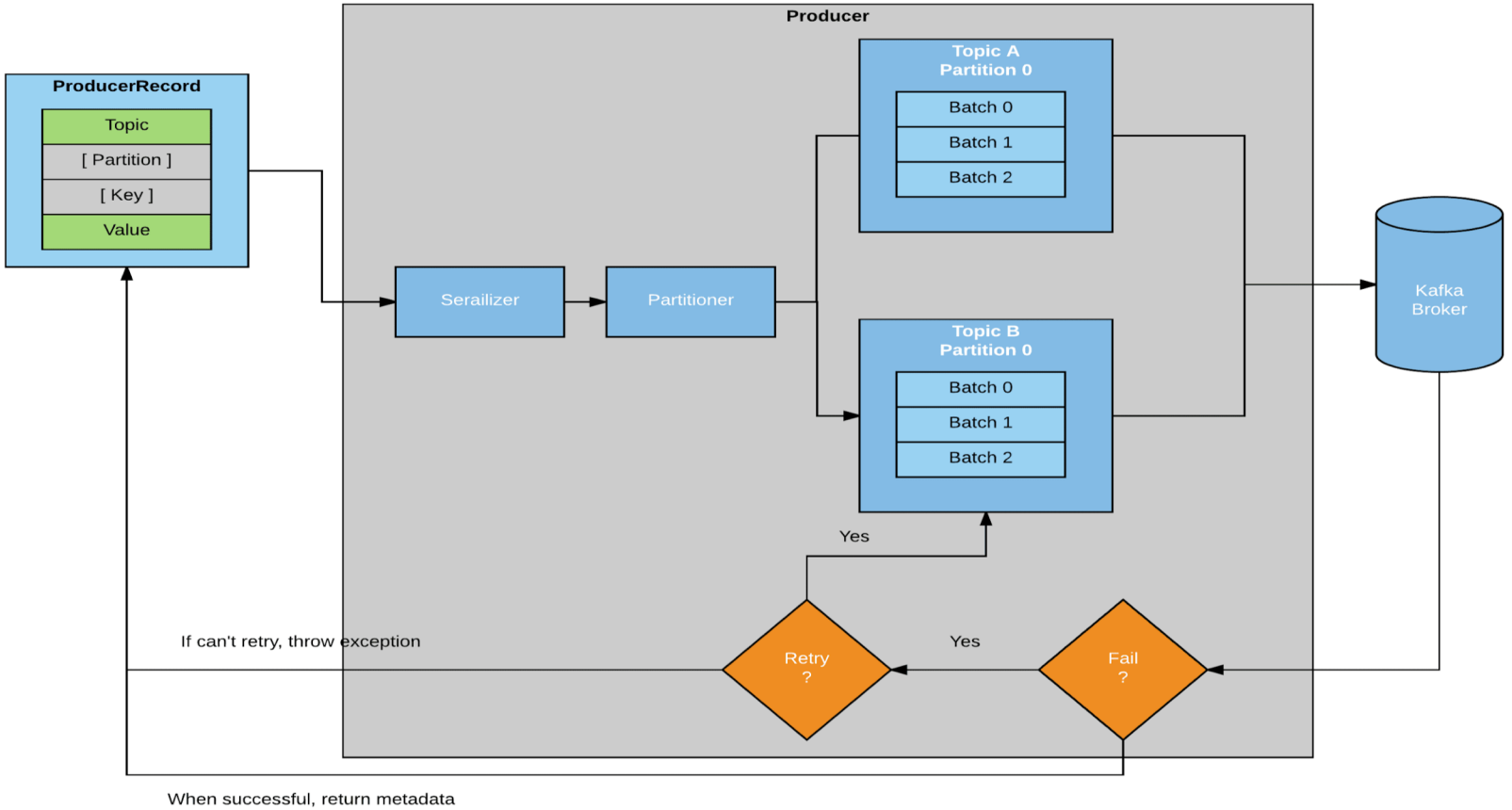
Producer API

- Different use case requirements will influence the way the producer API is used to write messages to Kafka and its configuration
- Three primary ways of sending messages
  - Fire-and-forget – send a message and don't really care if it arrived successfully or not. Most of the time it will arrive successfully but it's possible that some messages will get lost
  - Synchronous send – message is sent and a Future object is returned which can be used to see if the send() was successful
  - Asynchronous send – the send() method has a callback function which is triggered when a response is received from the Kafka broker

Acknowledgement of Messages

- No ack (0)
  - Kafka will most likely receive the message
  - Producer will not wait for any reply from the broker before assuming the message was sent successfully
  - If something goes wrong, producer will not know and message is lost
  - Because producer is not waiting for a response, high throughput can be achieved
- Ack from N replicas (1..N)
  - A message is not considered consumed by the Kafka cluster unless N replicas holding the message has acknowledged
- Ack from all replicas (-1)
  - Every replica must acknowledge the message

Producer Overview



Serialization

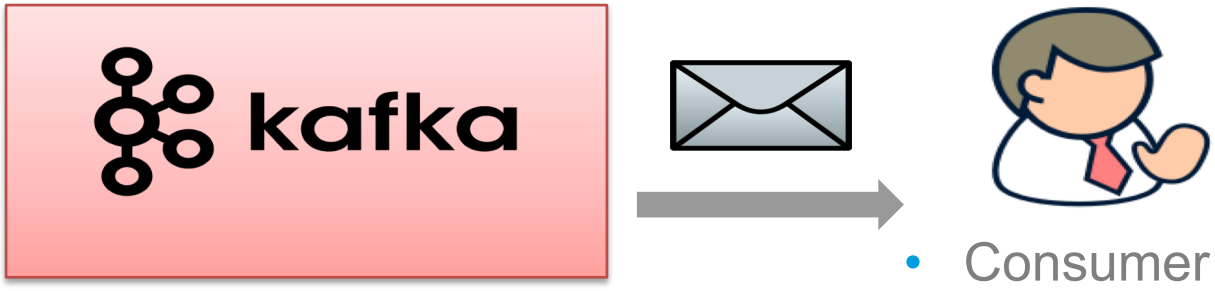
- Kafka messages are byte arrays (to Kafka)
  - Key → Array of bytes
  - Value → Array of bytes
- The Java API allows you to pass any object as key or value
  - Makes the code readable, but...
  - ... requires serializes and deserializes

- Kafka includes an interface for this
  org.apache.kafka.common.serialization.Serializer

Built-in Serializers

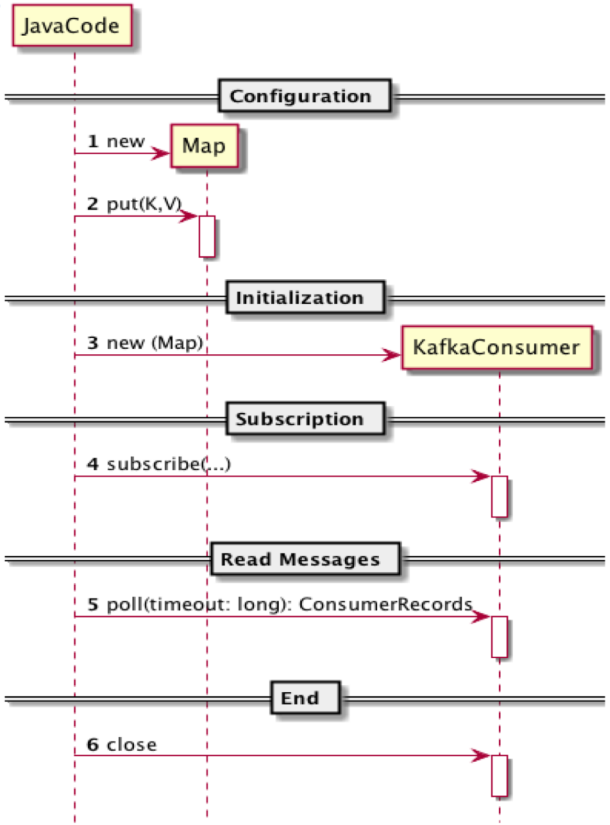
- Kafka includes serializes for common types:
  - ByteArraySerializer
  - StringSerializer
  - IntegerSerializer
  - …
- Most organizations settle on some standard serialization strategy
  - JSON, XML, Apache Avro, Protobuf

Consumers and Consumer Groups



- Applications that read data from Kafka are consumers
  - Subscribes to topics
  - Use KafkaConsumer to read messages from these topics
  - Kafka consumers are usually part of a *consumer group*
  - **The main way consumption of data from a Kafka topic is scaled is by adding more consumers to a consumer group**

Java API Behavior




```
// Configuration
Properties kp = new Properties();
kp.put("bootstrap.servers",
                "mybroker1:9092,mybroker2:9092");
kp.put("key.deserializer", "...");


// Initialization
KafkaConsumer<...> consumer=
                new KafkaConsumer<...>(kp);


// Subscription
consumer.subscribe("interesting.*");


// Read messages
ConsumerRecords<...> records = consumer.poll(100);
for (ConsumerRecord<...> cr : records) {
                // cr.value(); cr.key(); cr.offset();
}

// End
consumer.close()
```

Ballista

The Java API

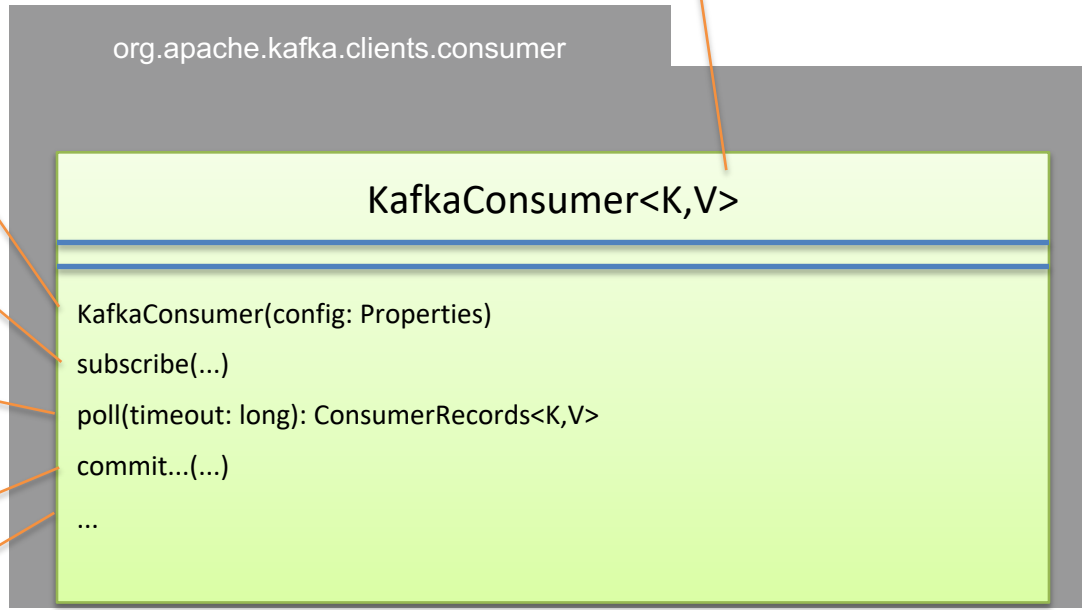
Generic sender where:
K = Type of key
V = Type of message

Constructor takes a configuration (mostly a hashmap of options)

Multiple ways to subscribe. Subscribe by list, wildcard, etc.

Read messages from Kafka

Multiple (sync/async) ways to confirm reception to consumer groups



A set of other methods such as: metrics(), pause(...), assign(...), close(), etc

Ballista

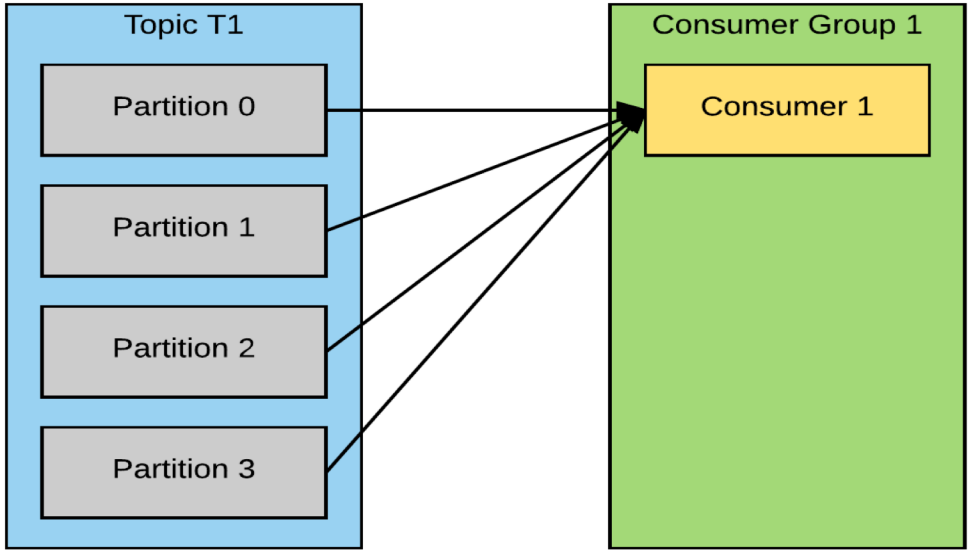
How to Use the API?

- The ***org.apache.kafka.clients.consumer.KafkaConsumer*** acts as a proxy for the consumer

- Some key issues to to resolve
  - Setup of consumer groups
  - Which topics to subscribe to
  - Which partitions to subscribe to (optional)
  - Manual or automatic offset management
  - Multi-threaded or single-threaded consumption

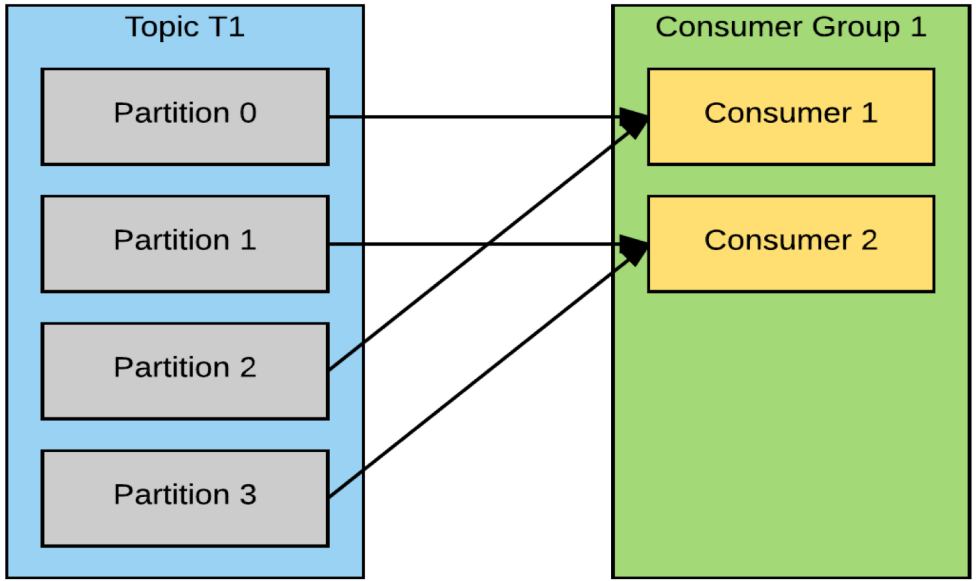
Consumer Group

- The consumer will receive all messages from all four partitions



Consumer Group

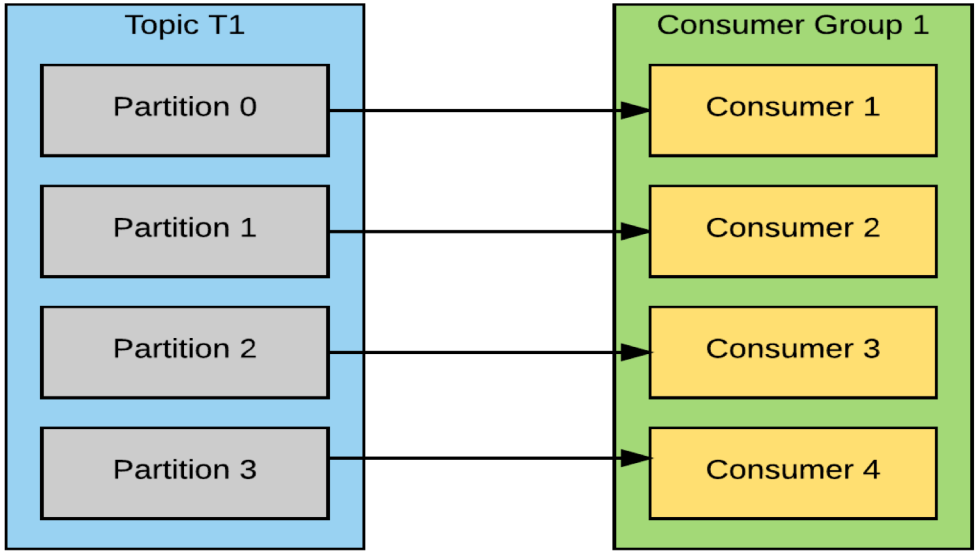
- Each consumer will only get messages from two partitions



Ballista

Consumer Group

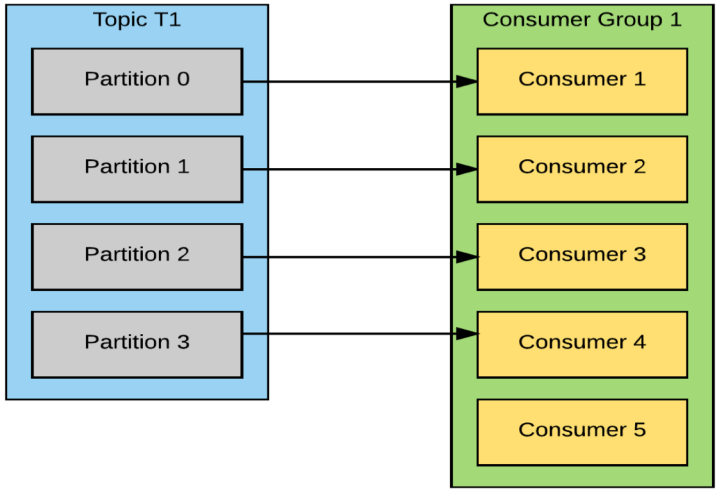
- If the consumer group has the same number of consumers as partitions, each will read messages from a single partition



Ballista

Consumer Group

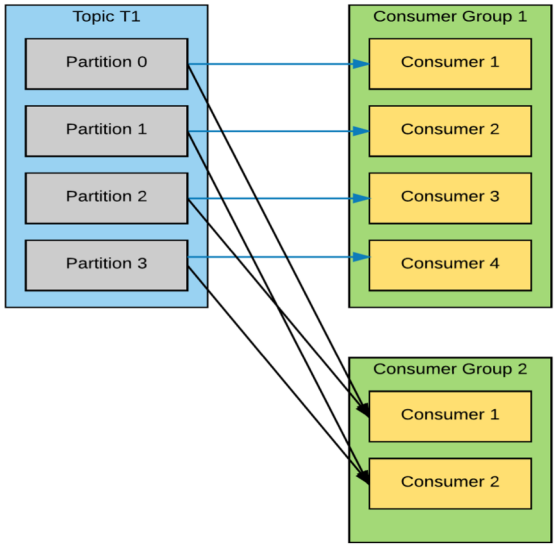
- If there are more consumers than partitions for a topic, some consumers will be idle and receive no messages
- Create topics with a large number of partitions to allow adding more consumers when load increases



Ballista

Multiple Consumer Groups

- One of the main design goals of Kafka was to allow multiple applications the ability to read data from the same topic
- Make sure each application has its own consumer group for this purpose



Ballista

Partition Rebalance

- Consumers in a consumer group share ownership of the partitions in the topics they subscribe to
- When a new consumer is **added** to the group, it consumes messages from partitions which were previously consumed by another consumer
- When a consumer **leaves** the group (shuts down, crashes, etc), the partitions it used to consume will be consumed by one of the remaining consumers
- Reassignment of partitions to consumers can also happen when topics are modified – an administrator adds new partitions, for example

Partition Rebalance

- The event in which partition ownership is moved from one consumer to another is called a *rebalance*
- During a rebalance, consumers can't consume messages!
- Steps can be taken to safely handle rebalances and avoid unnecessary rebalances

Creating a Kafka Consumer

- Creating a KafkaConsumer is similar to creating a KafkaProducer
- Like the producer, you must specify bootstrap.servers, key.deserializer, and value.deserializer in a Properties object
- You must also specify a group.id which specifies the consumer group for which the KafkaConsumer instance belongs to

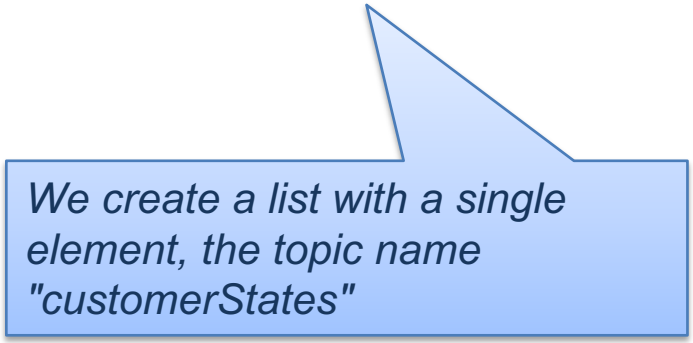
Setup of KafkaConsumer Example

```
private Properties kafkaProps = new Properties();
kafkaProps.put("bootstrap.servers", "mybroker1:9092,mybroker2:9092");

kakfaProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
kakfaProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
kafkaProps.put("group.id", "StateCounter");

KafkaConsumer<String, String> consumer = new KafkaConsumer<String, String>(kafkaProps);
```

Subscribing to Topics

- Once a consumer is created, you can subscribe to one or more topics

```
consumer.subscribe(Collections.singletonList("customerStates"));
```



Ballista

Subscribing with Regular Expressions

- You can also subscribe to topics using regular expressions
- The expression can match multiple topic names
- If a new topic is created with a name that matches, a rebalance will happen and consumers will start consuming from the new topic
- Useful for applications that need to consume from multiple topics
- Example: subscribe to all test topics
  - consumer.subscribe("test.*");

Consumer Poll Loop

- Once a consumer subscribes to topics, a loop polls the server for more data

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
  ConsumerRecords<String, String> records = consumer.poll(100);

  for (ConsumerRecord<String, String> record : records)
    System.out.printf("offset = %d, key = %s, value = %s", record.offset(), record.key(), record.value());
} finally {
  consumer.close();
}
```

Ballista

Consumer Poll Loop

- Once a consumer subscribes to topics, a loop polls the server for more data

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
  ConsumerRecords<String, String> records = consumer.poll(100);

  for (ConsumerRecord<String, String> record : records)
    System.out.printf("offset = %d, key = %s, value = %s", record.offset(), re                cord.value());
} finally {
  consumer.close();
}
```

Consumers must keep polling Kafka or they will be considered dead and the partitions they are consuming will be handed to another consumer in the group

Ballista

Poll Method

- poll() returns a list of records that contain:
  - Topic and partition the record came from
  - Offset of the record within the partition
  - Key and value of the record

- Takes a timeout parameter that specifies how long it will take to return, with or without data
  - *How fast do you want to return control to the thread that does the polling?*

Multithreading Considerations

- **One consumer per thread**
- To run multiple consumers in the same group in one application, each must run in its own thread
- You can wrap the consumer logic in its own object and then use Java's ExecutorService to start multiple threads each with its own consumer

Commits

- Unlike other JMS queues, Kafka does not track acknowledgements from consumers
- When poll() is called, it returns records that consumers in the group have not yet read

- The records that have been read by a consumer of the group are tracked by their position (offset) in each partition

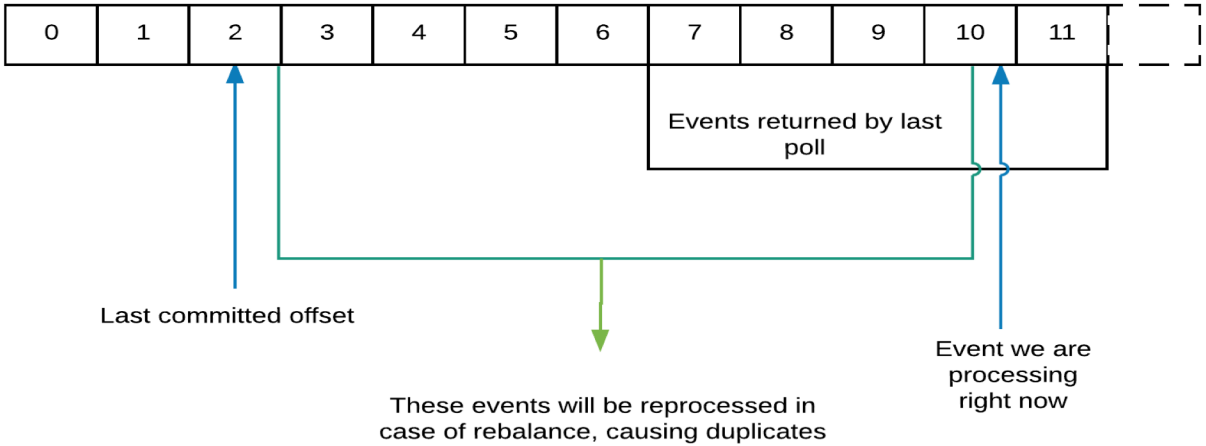
- When the current position in the partition is updated, it is known as a *commit*

Consumers Commit Offsets

- Consumers send a message to Kafka to a reserved topic with the committed offset for each partition
- If a consumer crashes or a new consumer joins the consumer group, a rebalance is triggered
- After a rebalance, a consumer may be assigned a new set of partitions and must read the latest committed offset of each partition and continue from there

Messages Processed Twice

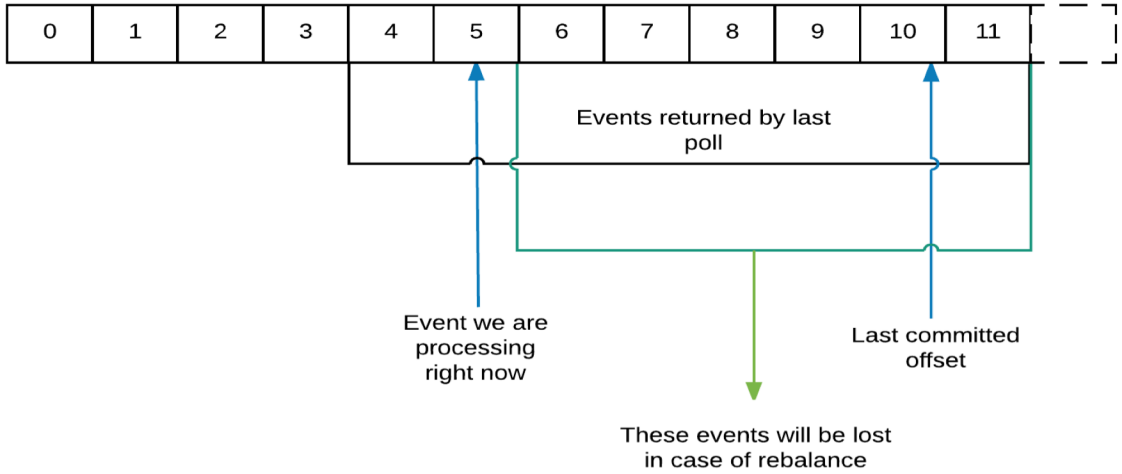
- If the committed offset is smaller than the offset of the last message the client processed, the messages between the last processed offset and the committed offset will be processed twice



Ballista

Messages Missed

- If the committed offset is larger than the offset of the last message the client processed, messages between the last processed and the committed offset will be missed by the consumer group



Ballista

Automatic Commit

- If you configure enable.auto.commit=true, the consumer will commit the largest offset your client received from poll() every 5 seconds by default
- Whenever there is a poll, the consumer checks if its time to commit and if so, will commit the offsets it returned in the last poll

- Convenient but don't give developers enough control to avoid duplicate messages

Commit Current Offset

- Developers usually want to exercise control over the time offsets are committed to eliminate possibility of missing messages **and** reduce the number of duplicate messages during rebalancing

- Setting `auto.commit.offset=false` means that offsets will only be committed explicitly

- Consumer has a `commitSync()` API to commit the latest offset returned by `poll()`

commitSync Example

```
while (true) {
  ConsumerRecords<String, String> records = consumer.poll(100);

  for (ConsumerRecord<String, String> record : records)
    System.out.printf("offset = %d, key = %s, value = %s", record.offset(), record.key(), record.value());

  try {
    consumer.commitSync();
  } catch (CommitFailedException e) {
    log.error("commit failed", e);
  }

} finally {
  consumer.close();
}
```

Once we are done processing all records in the current batch, commitSync is called before polling for more

Ballista

Asynchronous Commit

- The application is blocked until the broker responds to the commit request with commitSync() limiting throughput
- An alternative is to use commitAsync() which commits the last offsets and continues

```
while (true) {
  ConsumerRecords<String, String> records = consumer.poll(100);

  for (ConsumerRecord<String, String> record : records)
    System.out.printf("offset = %d, key = %s, value = %s", record.offset(), record.key(), record.value());

  consumer.commitAsync();
}
```

You can also pass a callback which is invoked by the consumer when the commit finishes (either successfully or not)

Ballista

Summary

- Producers
  - API
  - Sending messages
  - Serialization
- Consumers
  - API
  - Subscribing
  - Consumer groups
  - Rebalance

Ballista

Lab/Demo

- In this lab we'll use the Java API to consume and produce messages
- Two projects
  - Producer
  - Consumer
- Required tools:
  - Java
  - Maven
  - Docker (as before)
- Maven can be run stand alone or using a docker-container

Designs of Topics and Partitions

Ballista
TECHNOLOGY GROUP

Topic Design

- Name
- Schema
- Payload (Data)
- Key
- Number of partitions
- Number of replicas

The Short Story

- DevOps concerns
  - Bandwidth consumption: Size of messages, `serdes`, etc.
  - Fault tolerance and availability: Size of cluster, replication factor, etc.
  - Performance: Partitions, message size, cost of serialization, etc.
- Producer concerns
  - Ease of production: Clear schema, cost of serialization, delivery guarantees, etc.
- Consumer concerns
  - Can I subscribe to only what I need: Topics and partitions
  - Latency: Cluster size, performance, etc.

How Topics and Partitions Influence Concerns?

- Topic topology
  - Schema (structure, format, etc.)
  - Temporal constraints (frequency, triggers, etc.)
  - Do you use topics to allow for fine grain subscriptions?
- Partitions
  - Determines throughput (but not without cost)
  - Can be used for fine-grained subscription (requires use of low level API)
- Recommendation
  - Use topics to convey semantics
  - Use partition to control throughput

Name

- Descriptive name
- Evaluate use of hierarchy in naming
- Rule of thumb:
  - Use a longer name that is easy to understand

Schema

- JSON
  - Common choice
  - Not the most efficient

- Apache Avro
  - Binary format
  - Compression
  - Schema evolution
  - Dynamic typing (no code generation needed for serialization)

Partitions and Throughput

- Unit of parallelism: topic partition
- Writes to different partitions done in parallel
- Consumer: one thread get a single partitions data
- Consumer parallelism: bounded by the number of consumed partitions
- Throughput on a producer is a function of:
  - Batching size
  - Compression codec
  - Acknowledgement type
  - Replication factor
- Consumer throughput is a function of the message processing logic

Overpartitioning

- Problem: Increasing the number of partition and message ordering
  - If messages have keys, increasing the number of partitions may cause problems
  - Kafka maps a message to a partition based on the hash of the key
  - Messages with the same key go to the same partition
  - If we increase the number of partitions, this does not hold
    - Messages with the same key may for the retention period appear in multiple partitions
- Therefore:
  - Overpartition for a situation you expect in future

Too Many Partitions

- Each partition maps to a directory in the broker
  - 2 files: index, actual data
- You may need to configure the open file handle limit
  - Configuration
  - Seen in production > 30K open file handles / broker

Partitions and Availability

- Intra-cluster replication
- A partition can have multiple replicas, each on a different broker
- Clean broker shutdown: the controller moves the leaders off the broker that is shutting down
  - Takes a couple of ms
- Unclean shutdown: the loss of availability is dependent on the number of partitions
  - All replicas become unavailable at the same time
  - A new leader must be elected
  - Potential unavailability in seconds

Partitions and Client Memory

- A producer can set the amount of memory for buffering messages
  - Messages are buffered per partition
  - When buffer is full, messages are sent to the broker
- More partitions: more message buffering in the producer
- If out of memory, producer will block or drop new messages
  - Reconfigure
- Allocate at least a few tens of KB per partition

Summary

- Design topics based on your application semantics
  - Message types
  - Consumer concerns
  - Subscription granularity
- Decide on the number of partitions based on throughput
  - The more partitions, the higher theoretical throughput
  - Not without penalty
    - File handlers, consumer memory, latency, etc
  - Evaluate to overpartition to accommodate future growth

Lab: Design

- Let's assume you'll have to collect information from devices installed to keep track of the vitals of a set of patients
  - The patients are being treated from their home
  - You may assume that the patients produce 10 messages per second on average
  - You may have up to 1 million patients being tracked at the same time
- Multiple stakeholders
  - Nurse: Keeps track of a number of patients
  - Service technician: Need to know when a device goes down
  - Billing: Keeps track of events that are billable
  - ...

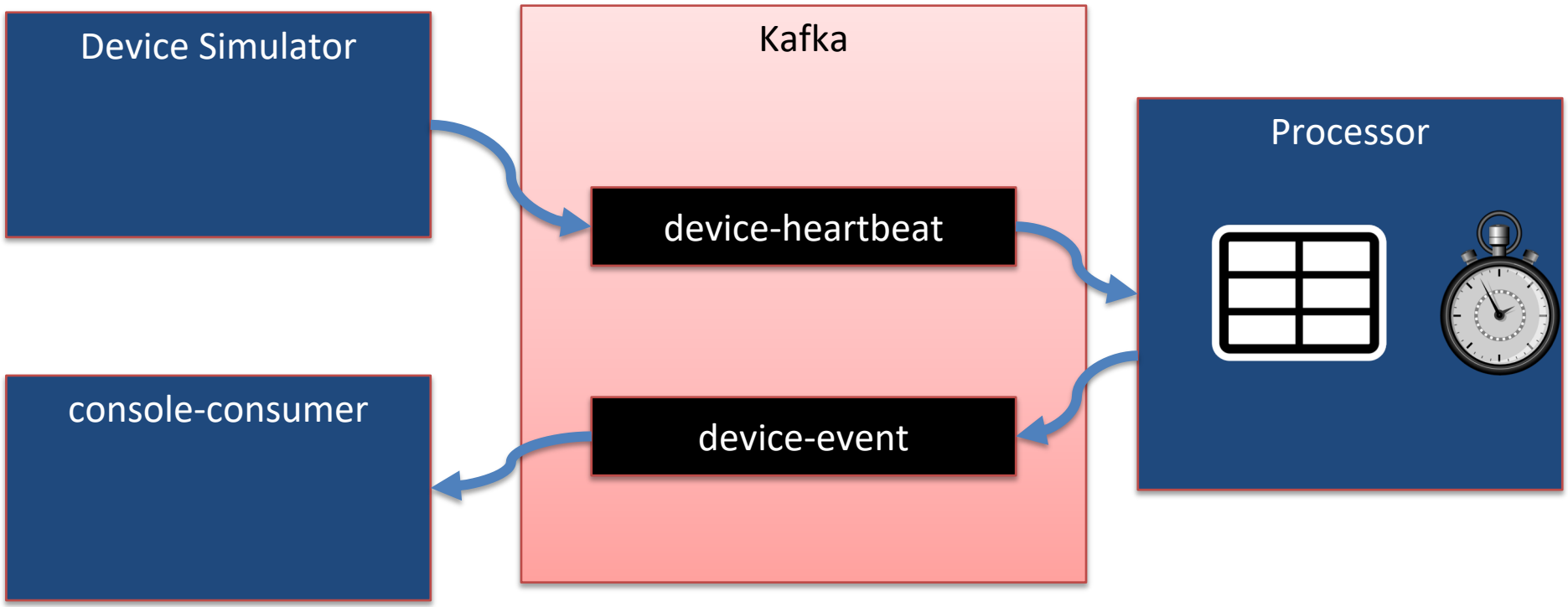
Lab: Design... The Question

- What would be the topics for such a system?
- How do we decide how many partitions per topic?

Lab: Implementation

- We'll take a look at a small sleeve of the problem
- Service technician's view:
  - Assuming all devices send a heartbeat
  - We want to know if the devices go offline and when they come back online
- We'll implement a processor that listens to the incoming heartbeats and decides if the device is online or offline

What We'll Run



Lab/Demo

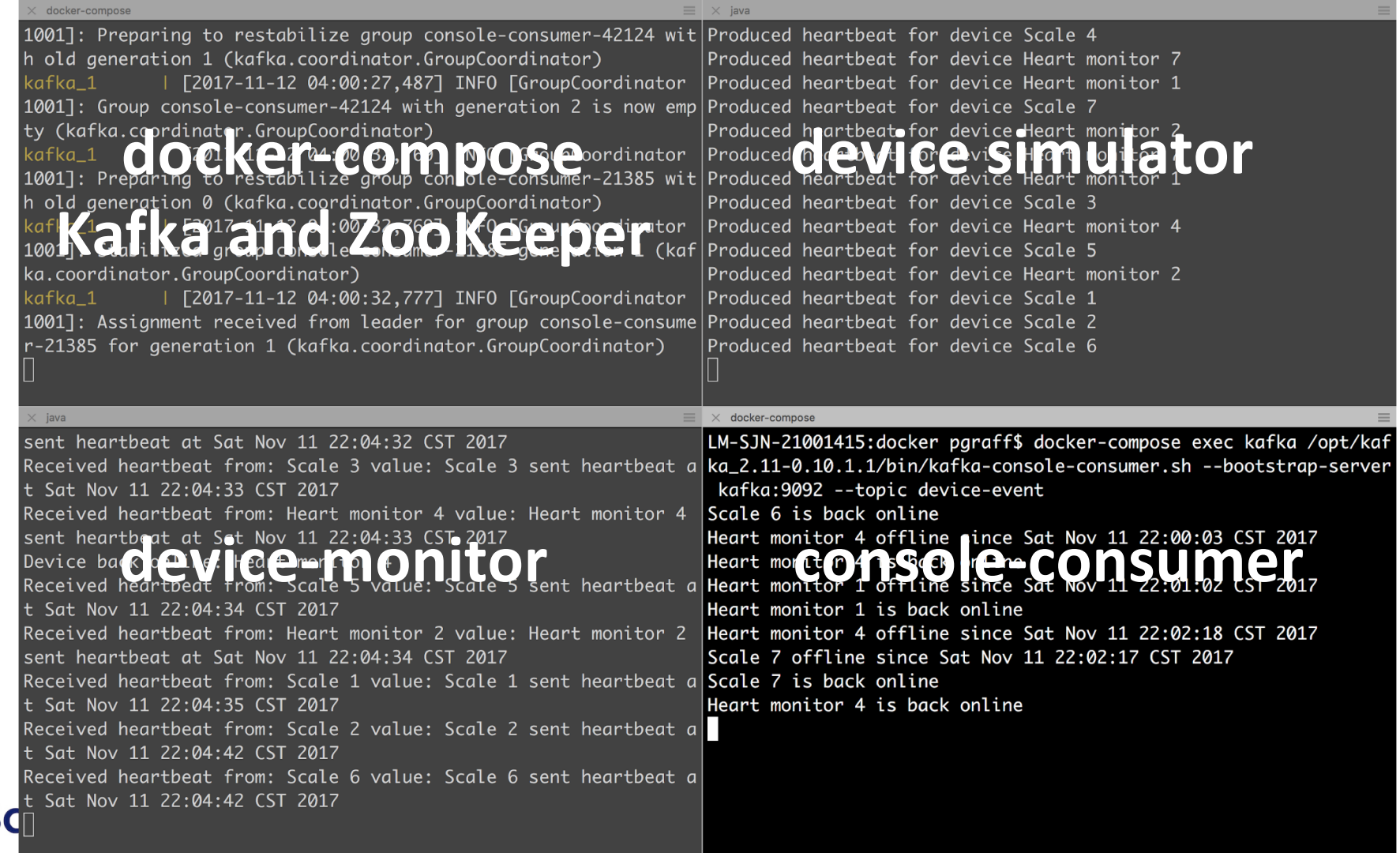
- Setup Kafka
  - Two topics
    - device-heartbeat
    - device-event
- Start the consumer(s)
  - At minimum, the listener to the device-event
- Build and run the processing daemon for the heart beats
  - Build in Java with Maven
- Build and run a simulator (simulating the devices)
  - Built in Java with Maven
- Observe the behavior of the application(s)

The Critical Code of DeviceSim

```
14   public DeviceSim(final String deviceId, final KafkaProducer<String, String> producer) {
15     timer.scheduleAtFixedRate(new TimerTask() {
16
17       @Override
18       public void run() {
19         if (r.nextBoolean()) {
20           System.out.println("Produced heartbeat for device " + deviceId);
21           producer.send(
22               new ProducerRecord<String, String> (
23                   "device-heartbeat",
24                   deviceId,
25                   deviceId + " sent heartbeat at " + new Date().toString()
26                   ));
27         }
28       }
29     }, r.nextInt(10000)+10000, 15*1000);
30   }
```

Interesting Code inside the Device Monitor

```
32      while (true) {
33        final ConsumerRecords<String, String> records = consumer.poll(1000);
34        for (ConsumerRecord<String, String> record : records) {
35          final String key = record.key();
36          lastSeenMap.put(record.key(), new Date());
37          System.out.println("Received heartbeat from: " + key + " value: " + record.value());
38          if (offlineDevices.contains(key)) {
39            offlineDevices.remove(key);
40            System.out.println("Device back online: " + key);
41            producer.send(createOnlineMessage(key));
42          }
43
44        }
45      }
```


docker-compose
Kafka and ZooKeeper
device simulator
device-monitor
console-consumer
1001]: Preparing to restabilize group console-consumer-42124 wit
h old generation 1 (kafka.coordinator.GroupCoordinator)
kafka_1 | [2017-11-12 04:00:27,487] INFO [GroupCoordinator
1001]: Group console-consumer-42124 with generation 2 is now emp
ty (kafka.coordinator.GroupCoordinator)
1001]: Preparing to restabilize group console-consumer-21385 wit
h old generation 0 (kafka.coordinator.GroupCoordinator)
ka.coordinator.GroupCoordinator)
kafka_1 | [2017-11-12 04:00:32,777] INFO [GroupCoordinator
1001]: Assignment received from leader for group console-consume
r-21385 for generation 1 (kafka.coordinator.GroupCoordinator)
Produced heartbeat for device Scale 4
Produced heartbeat for device Heart monitor 7
Produced heartbeat for device Heart monitor 1
Produced heartbeat for device Scale 7
Produced heartbeat for device Heart monitor 2
Produced heartbeat for device Heart monitor 1
Produced heartbeat for device Scale 3
Produced heartbeat for device Heart monitor 4
Produced heartbeat for device Scale 5
Produced heartbeat for device Heart monitor 2
Produced heartbeat for device Scale 1
Produced heartbeat for device Scale 2
Produced heartbeat for device Scale 6
sent heartbeat at Sat Nov 11 22:04:32 CST 2017
Received heartbeat from: Scale 3 value: Scale 3 sent heartbeat a
t Sat Nov 11 22:04:33 CST 2017
Received heartbeat from: Heart monitor 4 value: Heart monitor 4
sent heartbeat at Sat Nov 11 22:04:33 CST 2017
Received heartbeat from: Scale 5 value: Scale 5 sent heartbeat a
t Sat Nov 11 22:04:34 CST 2017
Received heartbeat from: Heart monitor 2 value: Heart monitor 2
sent heartbeat at Sat Nov 11 22:04:34 CST 2017
Received heartbeat from: Scale 1 value: Scale 1 sent heartbeat a
t Sat Nov 11 22:04:35 CST 2017
Received heartbeat from: Scale 2 value: Scale 2 sent heartbeat a
t Sat Nov 11 22:04:42 CST 2017
Received heartbeat from: Scale 6 value: Scale 6 sent heartbeat a
t Sat Nov 11 22:04:42 CST 2017
LM-SJN-21001415:docker pgraff$ docker-compose exec kafka /opt/kaf
ka_2.11-0.10.1.1/bin/kafka-console-consumer.sh --bootstrap-server
kafka:9092 --topic device-event
Scale 6 is back online
Heart monitor 4 offline since Sat Nov 11 22:00:03 CST 2017
Heart monitor 1 offline since Sat Nov 11 22:01:02 CST 2017
Heart monitor 1 is back online
Heart monitor 4 offline since Sat Nov 11 22:02:18 CST 2017
Scale 7 offline since Sat Nov 11 22:02:17 CST 2017
Scale 7 is back online
Heart monitor 4 is back online

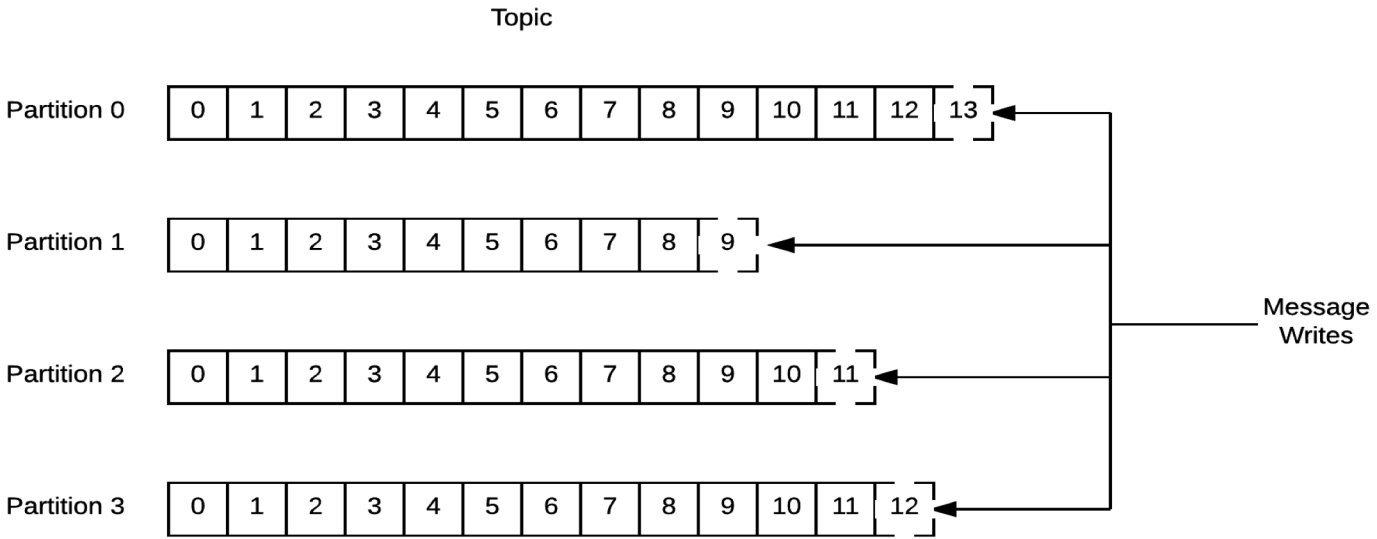
Scaling Kafka

Ballista
TECHNOLOGY GROUP

Topics and Partitions

- Messages in Kafka are categorized into *topics*
- Think of a topic as a database table or folder in a filesystem
- Topics are broken down into a number of *partitions*
- A topic generally has multiple partitions

Topics and Partitions



Ballista

Consumer Groups

- Consumers work as part of a *consumer group*
- One or more consumers that work together to consume a topic
- Group assures that each partition is only consumed by one member
- Mapping of a consumer to a partition is called *ownership* of the partition by the consumer