Apache Kafka - Real-time Data Processing

Nodar Momtselidze*
Ana Tsitsagi**

Abstract

Apache Kafka is creating a lot of buzz these days. While LinkedIn, where Kafka was founded, is the most well known user, there are many companies that use this technology successfully. Kafka has several features that make it a good fit for companies' requirements: scalability, data partitioning, low latency, and the ability to handle large number of diverse consumers. It works with Apache Storm and Apache Spark for real-time analysis and rendering of streaming data. The combination of messaging and processing technologies enables stream processing at linear scale. Common use cases include: Messaging, Website activity tracking, Log aggregation, Stream Processing, Commit log.

Keywords: Messaging, Website activity tracking, Log aggregation, Stream Processing.

Introduction

Kafka is one of those systems that is very simple to describe at a high level but has an incredible depth of technical detail when you look into it deeper. The Kafka documentation does an excellent job of explaining the many designs and implementation subtleties in the system. Kafka is a distributed, partitioned, and replicated commit log service. It provides the functionality of a messaging system but with a unique design. In this article, we will begin by briefly introducing Kafka, and then demonstrate some of Kafka's unique features by walking through an example scenario. Wewill also cover some additional usage cases and also compare Kafka to existing solutions.

Real Time Data Processing Challenges

Real Time data processing challenges are very complex. As we all know, Big Data is commonly categorized into volume, velocity, and variety of the data, and Hadoop like system handles the Volume and Varity part of it. Along with the volume and variety, the real time system needs to handle the velocity of the data as well. And handling the velocity of Big Data is not an easy task. First, the system should be able to collect the data generated by real time events streams coming in at a rate of millions of events per seconds. Second, it needs to handle the parallel processing of this data as and when it is being collected. Third, it should perform event correlation using a Complex Event Processing engine to extract the meaningful information from this moving stream. These three steps should happen in a fault tolerant and distributed way.The real time system should be a low latency system so that the computation can happen very fast with near real time response capabilities.

Figure 1:

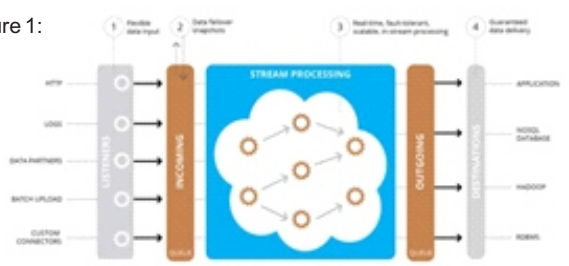


Figure 1 depicts a different construct of a real time system. Streaming data can be collected from various sources, processed in the stream processing engine, and then write the result to destination systems. In between, the Queues are used for storing/buffering the messages.

How Kafka works

Like many publish-subscribe messaging systems, Kafka maintains feeds of messages in topics. Producers write data to topics and consumers read from topics. Since Kafka is a distributed system, topics are partitioned and replicated across multiple nodes. Messages are simply byte arrays and the developers can use them to store any object in any format – with String, JSON, and Avro, the most common one. It is possible to attach a key to each message, in that case the producer guarantees that all messages with the same key will arrive to the same partition. When consuming from a topic, it is possible to configure a consumer group with multiple consumers. Each consumer in a consumer group will read messages from a unique subset of partitions in each topic they subscribe to, so each message is delivered to one consumer in the group, and all messages with the same key arrive at the same consumer. What makes Kafka unique is that it treats each topic partition as a log (an ordered set of messages). Each message in a partition is assigned a unique offset. Kafka does not attempt to track which messages were read by each consumer and only retain unread messages; rather, Kafka retains all messages for a setamount of time, and consumers are responsible to track their location in each log. Consequently, Kafka can support a large number of consumers and retain large amounts of data with very little overhead.

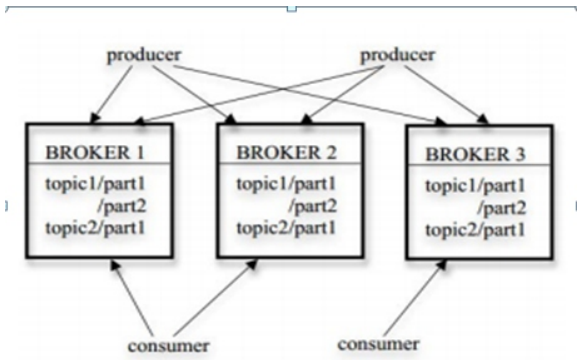
Massaging system

Apache Kafka is the messaging system originally developed at LinkedIn for processing LinkedIn's activity stream.

*Prof. Nodar Momtselidze Professor at the Faculty of Computer Technologies and Engineering
International Black Sea University, E-mail: nmomtselidze@ibsu.edu.ge

Journal of Technical Science and Technologies; ISSN 2298-0032; e-ISSN 2346-8270; Volume 4, Issue 2, 2015

Figure 2:



The overall architecture of Kafka is shown in Figure 2. Kafka is distributed in nature which typically consists of multiplebrokers. To balance load, a topic is divided into multiple partitions and each broker stores one or more of those partitions. Multiple producers and consumers can publish and retrieve messages at the same time. Kafka relies heavily on the file system for storing and caching messages. There is a general perception that "disks are slow" which makes people skeptical that a persistent structure can offer competitive performance. In fact, disks are both much slower and much faster than people expect depending on how they are used; a properly designed disk structure can often be as fast as the network. The key fact about disk performance is that the through put of hard drives has been diverging from the latency of a disk seek for the last decade. As a result, the performance of linear writes on a 6 7200rpm SATA RAID-5 where array is about 300MB/sec but the performance of random writes is only about 50k/sec withdifferences of nearly 10000X! These linear reads and writes are the most predictable of all usage patterns, and hence the one detected and optimized best by the operating system using read-ahead and write-behind techniques. Kafka has a very simple storage layout. Each partition of a topic corresponds to a logical log.

Physically, a log is implemented as a set of segment files of approximately the same size (e.g., 1GB). Every time a producer publishes a message to a partition, the broker simply appends the message to the last segment file. For better performance, Kafka flush the segment files to disk only after a configurable number of messages have been published or a certain amount of time has elapsed. A message is only exposed to the consumers after it is flushed.

Figure 3:

Figure 3 shows how different types of producers can communicate to different types of consumers through the Kafka Broker.Kafka is explicitly distributed—producers, consumers, and brokers can all be run on a cluster of machines that co-operate as a logical group.This happens fairly naturally for brokers and producers, but consumers require particular support. Each consumer process belongs to a consumer group and each message is delivered to exactly one process within every consumer group. Hence, a consumer group allows many processes or machines to act logically as a single consumer. The concept of a consumer group is very powerful and can be used to support the semantics of either a queue or topic as found in JMS. To support queue semantics, we can put all consumers in a single consumer group, in which case each message will go to a single consumer. To support topic semantics, each consumer is put in its own consumer group, and then all consumers will receive each message. Kafka has the added benefit in the case of large data that no matter how many consumers a topic has, a message is stored only a single time.

Kafka components

1. Quick start: install vagrant, install virtual box, git clone, cd skala-kafka, vagrant up. Zookeeper should be running.

2. Creating producers:

```
val producer = new KafkaProducer("test-topic","192.168.86.10:9092")
producer.send("hello distributed commit log")

case class KafkaProducer( topic: String, brokerList: String,
clientId: String = UUID.randomUUID().toString,
synchronously: Boolean = true, compress: Boolean = true,
batchSize: Integer = 200, messageSendMaxRetries: Integer = 3,
requestRequiredAcks: Integer = -1)

val props = new Properties()
val codec = if(compress) DefaultCompressionCodec.codec
else NoCompressionCodec.codec
props.put("compression.codec", codec.toString)
props.put("require.requred.acks",requestRequiredAcks.toString)
val producer = new Producer[AnyRef, AnyRef](new ProducerConfig(props))
def kafkaMesssage(message: Array[Byte],
partition: Array[Byte]): KeyedMessage[AnyRef, AnyRef] = {
if (partition == null) {
new KeyedMessage(topic,message)
} else {new KeyedMessage(topic,message, partition)}
}

def send(message: String, partition: String = null): Unit = {
send(message.getBytes("UTF8"), if (partition == null) null
else partition.getBytes("UTF8"))}
def send(message: Array[Byte], partition: Array[Byte]): Unit = {
try {producer.send(kafkaMesssage(message, partition))}
catch {case e: Exception =>e.printStackTrace
System.exit(1)} }
```

3. Creating consumers:

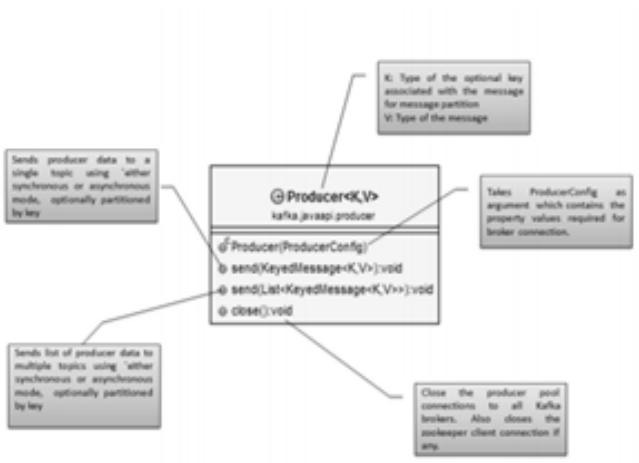
Journal of Technical Science and Technologies; ISSN 2298-0032; e-ISSN 2346-8270; Volume 4, Issue 2, 2015

```
class KafkaConsumer(topic: String, groupId: String,
zookeeperConnect: String)

val props = new Properties()
props.put("group.id", groupId)
props.put("zookeeper.connect", zookeeperConnect)
props.put("auto.offset.reset",
if(readFromStartOfStream) "smallest" else "largest")
val config = new ConsumerConfig(props)
val connector = Consumer.create(config)
val filterSpec = new Whitelist(topic)
val stream = connector.createMessageStreamsByFilter(filterSpec, 1,
new DefaultDecoder(), new DefaultDecoder()).get(0)

def read(write: (Array[Byte])=>Unit) = {
for(messageAndTopic <- stream) {
try {write(messageAndTopic.message)}
catch {case e: Throwable =>
error("Error processing message, skipping this message: ", e)}
} }

val consumer = new KafkaConsumer("test-topic","groupTest",
"192.168.86.5:2181")
```

The Java Producer API

Producer: Kafka provides the Producer class (class Producer<K,V>) for creating messages for single or multiple topics with message partition as an optional feature. The following is the class diagram and its explanation:

Figure 4:



Here, Producer is a type of Java generic written in Scala where we need to specify the type of parameters; K and value, respectively. Keyed Message: The Keyed Message class takes the topic name, partition key, and the message value that needs to be passed from the producer as follows: class Keyed Message [K, V] (valtopic: String, val key: K, val message: V) Here, Keyed Message is a type of Java generic written in Scala where we need to specify the type of the parameters; K and V specify the type for the partition key and message value, respectively, and the topic is always of type String. Producer Config: The Producer Config class encapsulates the values required for establishing the connection with brokers such as the broker list, message partition class, serializer class for the message, and partition key.

This Simple Producer class is used to create a message for a specific topic and transmit it. As the first step, we need to import the following classes:

```
import kafka.javaapi.producer.Producer;
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;
```

As the next step in writing the producer, we need to define properties for makinga connection with Kafka broker and pass these properties to the Kafka producer:

```
Properties props = new Properties();
props.put("metadata.broker.list","localhost:9092");
props.put("serializer.class","kafka.serializer.StringEncoder");
props.put("request.required.acks", "1");
ProducerConfig config = new ProducerConfig(props);
Producer<Integer, String> producer = new Producer<Integer,
  String>(config);
```

As the final step, we need to build the message and send it to the broker as shown in the following code:

```
package test.kafka;

import java.util.Properties;
import kafka.javaapi.producer.Producer;
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;

public class SimpleProducer {
    private static Producer<Integer, String> producer;
    private final Properties props = new Properties();
    public    SimpleProducer()
    {
      props.put("broker.list", "localhost:9092");
      props.put("serializer.class", "kafka.serializer.StringEncoder");
      props.put("request.required.acks", "1");
      producer = new Producer<Integer, String>(new
      ProducerConfig(props));
    }

 public static void main(String[] args) {
     SimpleProducer sp = new SimpleProducer();
     String topic = (String) args[0];
     String messageStr = (String) args[1];
     KeyedMessage<Integer, String> data = new KeyedMessage<Integer,
     String>(topic, messageStr);
     producer.send(data);
     producer.close();
   }
 }
```

Compile the preceding program and use the following command to run it: [root@localhost kafka-0.8]# java Simple Producer kafka topic Hello_There_Here, kafka topic is the topic that will be created automatically when the message Hello_There is sent to the broker.
Conclusion

References

Garg, N. (2013 October) Apache Kafka.
Mitra, M. (2013 EMC Proven Professional Knowledge Sharing).analytics on big fast data using real time stream data processing architecture.

Stein, J.(BDOSS). Apache Kafka, realtime data pipelines.
Kreps,J,; Narkhede, N., & Rao, J. (Kafka: a Distributed Messaging System for Log Processing.

Mitra, M. (2013 EMC Proven Professional Knowledge Sharing). analytics on big fast data using real time stream data processing architecture.

Barrera, P. Reliable RT processing Onnen.E.(September 27, 2012). Data Models and Consumer Idioms Using Apache Kafka for Continuous Data Stream Processing.