DEVELOPING MESSAGING AND REAL TIME PROCESSING SYSTEM FOR CLOUD CONNECTED CARS

By

GEORGIOS CHANTZIALEXIOU

A thesis submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

For the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Maria Striki

And approved by

New Brunswick, New Jersey

October, 2019

ABSTRACT OF THE THESIS

Developing Messaging and Real Time Processing System for Cloud Connected Cars

by Georgios Chantzialexiou

Thesis Director: Maria Striki

In the recent years, the interest in developing self driving cars, autonomous drones and connected cars skyrockets. That is leading to the need to develop a cloud messaging system with close to real time capabilities that enable vehicles share information to each other in order to help them improve their navigation. Although, there are many popular existing cloud messaging and processing solutions, these engines introduce diverse characteristics and runtime architectures, so there is a need to analyze not only the resources they require but also the execution time they manage to achieve. The complexity of the task, is also affected by execution parameters of the underlying algorithm. The outcome of such analysis will provide us with the means to understand the advantages and disadvantages of every execution engine under specific circumstances, and also let us deploy user policies in cloud environments that relate to the cost and the time restraints of the executions. For this purpose, we must conduct an experimental analysis on those engines through a profiling process, where we will measure the usage of the resources as well as the overall execution time. The results of this process will enable us to construct static predictive models that could simulate the performance of the engines for varying execution parameters.

In this thesis we used Kafka as our distributed messaging system and measured the

communication between cars. We choose Kafka instead of other messaging systems due to its reliability, scalability, ease of use, proven success, and popularity across the big data community.

Furthermore, we used Apache Spark as the real-time processing engine. We chose Spark because it is easy to integrate it with Kafka, for its scalability, reliability, ease of use and its popularity. Moreover, the Machine Learning library of Spark is widely used. In order to analyze the suitability of the above system we developed mini applications that simulate real-world scenarios to analyze the performance of the system. We run experiments using different settings and different workloads and measure performance that help us understand the behavior of the system.

Keywords: Apache Spark, Apache Kafka, Big Data, Autonomous Vehicles, Parallel Processing, Real Time Processing, Robotics

Acknowledgements

I would like to thank my supervisor Maria Striki, who helped me to realize this thesis. Her help was very important in carrying out this work. I am particularly pleased that the cooperation we have had throughout my studies will be closed with a final project thesis. I would also like to thank my family for supporting me from the day I was born. I would also like to thank Dr. Petropoulou with whom she helped me make my first steps in the research. Finally, I would like to thank my friends with whom I spent very beautiful years in the U.S.A. Without them, the graduate school would be very monotonous.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

An increasing number of vehicle related platforms are incorporating technologies that would benefit from communication between central and connected cars.

- **connected cars**: would benefit from the data sharing in order to obtain information that will improve their driving experience. For example, if all cars are sharing their location it would be easier to detect the traffic jam and avoid them. Another example would be collision avoidance. Cloud system could read the location and the trajectory of each car on the road and if two car trajectories are going to be met, which means that cars are going to crash, cloud will be able to warm well in advance the drivers to take action to avoid the crash. Moreover, cloud will be able to take control of the cars in order to avoid the crash. There are dozen of examples and usecases for connected cars that would benefit from a cloud-car communication.

- **self driving cars**: have a much broader range of usecases that would benefit from such a system. I am going to provide a few important examples: Self-driving car motion-planning, object detection algorithms can greatly improve from sharing the predictions and the input back to the cloud. Cloud algorithms will utilize their unlimited resource availability to improve their models. Collision avoidance is mandatory in these type of cars because they don't have real driver. Recently, a Tesla car crashed to another car which lead to a fatal accident as the Tesla car failed to detect the other car because of its white color. A cloud connected system would be able to alarm both cars using the gps about the upcoming collision and avoid the accident.

A cloud system is needed to establish communication between the cars. The system should have the following capabilities:

- Reliability: The system should provide fault tolerance in data transfer and computing. The system should be able to provide delivery guarantees.
- Availability: The system should have very high up-time. It shouldn't crash, or pause for maintenance
- Real time: For certain use-cases the system should be able to compute and provide information in real time or close to real time.
- Scalability: The system should be able to scale in order to accommodate for users. For example during peak hours.
- Security: Data should be safely transferred to and from the cloud to ensure no privacy invasion.

1.2 Contribution

In this work I propose the development and characterization of a streaming system that will be able to cover all the above requirements described at the introduction above.

As a messaging system I used Apache Kafka.
Kafka has the following claims:

- Security: Kafka is able to encrypt the data in order to safely transfer them.
- Scalability: Kafka is able to scale across many brokers without overhead thanks to the help of zookeeper which is coordinating the Kafka brokers.
- Reliability: Kafka is able to replicate data across brokers. If one broker fails, Zookeeper will make sure that another broker will take over as a new leader and

try to restart the failed broker. Zookeeper could be launched across many nodes too to make sure that one zookeeper node is not a single point of failure.

- Real-time: Kafka can send messages with high throughput and low latency.
- Availability: As previously said, Kafka can have many brokers, which leads to high availability. Each node could be paused for maintenance reasons and restarted.

As a processing engine I used Apache Spark. Spark claims that:

- Reliability: Kafka has implemented fault-tolerance using DAG (Direct Acyclic Graph). If a node fails, another node will take over the work of the bad node from the DAG and do it. It is also very efficient.
- Security: Data are read from Kafka[1] and kept in the cloud. Therefore, there is no high risk for data attack on Kafka.
- Scalability: Spark can easily scale across many cores/node. Correct configuration between Spark and Kafka will also improve parallelism.
- Availability: Spark;s architecture is master-slave, which means that if master node fails the whole system will fail. To overcome this issue Spark is able to spawn many master nodes. Therefore, chances of downtime are pretty low.
- Real time: Spark has developed a micro batch streaming system which is able to compute data in close to second window time. Certain operations could also be completed sub second.

Furthermore, I developed a few example applications that are representative of real world usecases in order to measure the performance of the system and verify that it is compliant with the aforementioned claims. Moreover, by developing the applications I got to assess the level of difficulty to develop, and the resources required to maintain streaming applications and system configurations.

1.3 How The Work Is Organized

- In chapter 2 : We present the general theoretical background, tools and applications associated with autonomous vehicles. The reader can familiarize themselves with the frameworks and technologies used in the current work. These are considered necessary for the study that will follow in subsequent work units.

- In chapter 3: We describe the system's design and implementation. We focus on the study and analysis of each of the software components of the system. We provide high-level overview of the system and its characteristics, followed by detailed information for its components, including the data generation and input part, the Kafka topic, spark streaming and Kafka - Spark connector.

- In chapter 4: We discuss applications that the system architecture will have to support.

- In chapter 5: We explain the Experimental setup and the experiments that I performed at the current project thesis and evaluate the performance of the system. Firstly we describe the datasets used. Next, we perform experiments to evaluate the performance and scalability of proposed system.

- In chapter 6: We summarize the conclusions drawn from my study and discuss possible future steps.

Chapter 2

Theoretical Background

2.1 Overview

We present the general theoretical background, tools and applications associated with autonomous vehicles. The reader can familiarize themselves with the frameworks and technologies used in the current work. These are considered necessary for the study that will follow in subsequent work units.

2.2 Messaging Systems

Messaging systems are all about data. Their goal is to transfer small and large amounts of data in a reliable rapid and scalable way[2]. Most of the usecases that need to transfer data require the messaging system to transfer large amounts of data fast (which means the system has to be High-Throughput), and usually they expect the system to send and receive messages in a reliable way. That means the system has to be fault-tolerant in order to meet the expectations of popular usecases.

What are the challenges in Messaging systems?
A message broker could become a bottleneck in the scalability of the system. Common issues that cause the broker to to lag is messages with big size. Big and small message size is relative term. Each message system consider a message big different than the others. For example for Kafka a message of 1 Mega Byte is considered big and it is not recommended by the designers of the system. The biggest problem with big message sizes is that the I/O of the filesystem that the broker is writing the data is not very high. The last few years, CPUs have keep growing very fast, but the disk speed has not followed the progress of the CPUs. As a result, sending many big messages usually

cause file-systems and therefore brokers to lag and become bottlenecks of the system. Other problem that a system could face is when an application that is producing data is using only one node. That could limit create both I/O issues similar to the one I previously described but also it could create a network issue, because the network bandwidth is limited within one host.

The same exact issue could happen if the consumer is slow. That happens usually due to bad network connection or again if the consumer is using only one node. Low data consumption could make the broker system flood with data and therefore fail. As a result, data could be lost and the system will not be able to receive or send data.

Usually, messaging systems are expected to enable the data subscribers reprocess data. There are multiple reasons that subscribers could need to reprocess data. Example: subscriber could fail at some point and loose all the received data, or a bug in the subscriber's logic could invalidate all the received data and therefore the only solution would be for the application to ask the data again from the messaging system. Of course, every feature comes with a cost. Saving large amount of data for offline processing means that the system requires large amount of disk space.

2.3 Zookeeper

2.3.1 Distributed File-Systems

The truth about cloud systems is that they have many file-systems. How cloud works? There are many physical hardware drives that could be interconnected to other machines using high performance network systems. As a result, cloud has many file systems that somehow should talk to each other in to develop distributed services. Many companies and academics have developed their own distributed file system. For example: The most popular distributed file system in academia is:

- Lustre[3], which is used by many supercomputers for large-scale computing.

- GFS[4]: Google file system which is developed and used by Google Inc.

- Azure[5]: which is developed by Microsoft Cloud Team.

How do these systems work?

These systems usually are usually the following architecture:

- They have A Name Node services, which should continuously run: to track meta-data file. There is a very similar concept in Linux kernel, the inode, which is responsible for creating/updating/deleting file information such as size, read/write permissions, time last created/updated.

- In the distributed system the name node should be able to tell you which node is holding your data.
- System has a node which is responsible for load balancing the files between nodes.

- System has a node which should be able to replicate data for backup (if needed).

So, if a user needs to access a file, he has to query the inode and them the inode will return the pointer that references that file. The benefit of this architecture is that:

- It will make no difference to the user for UI perspective.

- It is reliable (because it can replicate the data to many different nodes)

- Data shows that it able to successfully perform large reads/writes on large files.

On the other side:

- The master node is not able to handle many requests at once, especially if many users are asking to write to the same files. That could lead to a node failure or a denial of service.

- The NameNode if it is only one, will be a single point of failure, and a large machine with thousand of nodes will fail because of one node.

- File-system is not able to perform real time (from streaming) large scale I/O [6]

Could a files-system be used to keep track of a distributed messaging system like Kafka?

File-System have limitations that fail to meet the requirements of Kafka brokers. Let's consider the following approach:
Let's consider that we deployed a Kafka system with 5 brokers and wee need to write to a file the status of each broker such as: "uptime", "broker-crashed". What would happen if 2 nodes need to append data to the file at the same time? Some of the data will be lost or get corrupted because there is no easy way to lock the file descriptor and prevent the other file from writing data to it.
What is the root cause of this problem?
There is no detection of double writing attempts. One can't put a lock to the file. Even if that was true, it would cause other problems such as race conditions issues that would kill the performance of the system, where you could have many users.
In a single node system that wouldn't be an issue because it is easy to detect that more than one user has requested the file descriptor from the file-system, but not in distributed file systems. The main reason that we cannot add this feature to a distributed system is that there are other protocols that would need to be extended to

do that and they have pretty bad performance. Therefore, adding that wouldn't solve anything.

In order to solve these issue Zookeeper has been developed, which is a wait-free coordination for Internet-scale systems

2.3.2 Apache Zookeeper - Architecture

What is Zookeeper[7] It is a service for coordinating processes of distributed applications. Zookeeper is trying to provide simple and high performance kernel for building distributed applications that need complex coordination primitives at the client. It incorporates elements from group messaging, shared registers, and distributed lock services in a replicated, centralized service. The interface exposed by ZooKeeper has the wait-free aspects of shared registers with an event-driven mechanism similar to cache invalidation of distributed file systems to provide a simple, yet powerful coordination service. The ZooKeeper interface enables a high-performance service implementation. In addition to the wait-free property, ZooKeeper provides a per client guarantee of FIFO execution of requests and linearizability for all requests that change the ZooKeeper state. These design decisions enable the implementation of a high performance processing pipeline with read requests being satisfied by local servers. ZooKeeper can handle tens to hundreds of thousands of transactions per second. This performance allows ZooKeeper to be used extensively by client applications like Apache Kafka.
Where should I use Zookeeper?
By no means, Zookeeper should never be used to for storage instead of a file-system. Zookeeper is good at active communication. The strong elements of Zookeeper are the following:

- It is good at electing leaders. (for example in kafka, electing one broker as the leader)

- Group membership.

- Active configuration management.
- Status monitoring.
- Implementing distributed locks.

Zookeeper saves all the data on znode, znode is an in-memory data node in the zookeeper data. The system is following hierarchical node organization which is called data tree. The main reason for this organization choice was made for simplicity and commonality, because users are used to the tree logic from the filesystems. Most popular system that follows this structure is the UNIX system. Users can access data nodes the same way they would access files in UNIX. Below we can see an illustration of the hierarchical data tree structure of zookeeper znodes.

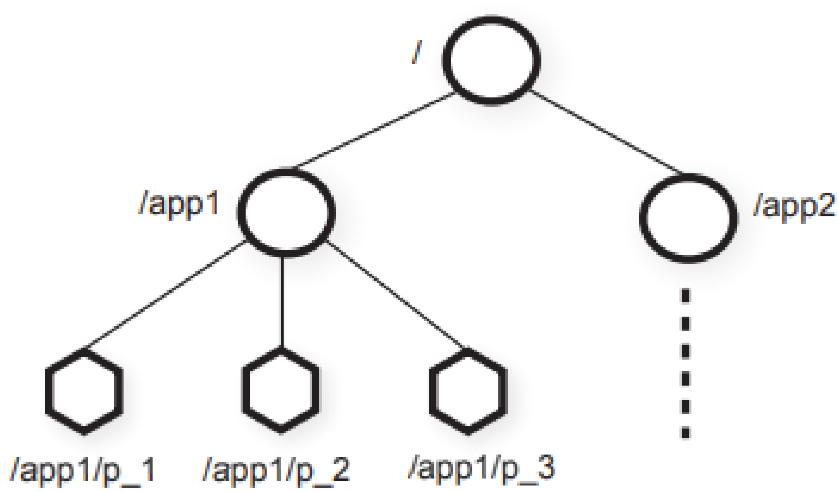

Figure 2.1: Zookeeper Znode Data Tree Structure

Sessions. A client connects to ZooKeeper and initiates a session. Sessions have an associated timeout. ZooKeeper considers a client faulty if it does not receive anything from its session for more than that timeout. A session ends when clients explicitly close a session handle or ZooKeeper detects that a clients is faulty. Within a session, a client observes a succession of state changes that reflect the execution of its operations. Sessions enable a client to move transparently from one server to another within a ZooKeeper ensemble, and hence persist across ZooKeeper servers

As far as znode is concerned, there are two different znode types:

- Regular: Clients can create and delete nodes explicitly.

- Ephemeral: Similar to regular znodes, but they are also linked with session. If a session expires then the Ephemeral node gets deleted.

  Both types have an increasing counter to their name.

How locks are implemented?

A) The leader would create a lock file. For example: leader/locker B) The leader would use an ephemeral node to save the file. Why is that? If the node dies or finishes it kills the session and then the node is automatically destroyed. As a result, other nodes that needed the lock would automatically watch this change and create another ephemeral node to acquire lock.

What are the limitations of Zookeeper?

Zookeeper is good at actively handling (reading, writing) events, so it can be used for active configuration of systems, but it does not by design persisting data. If the system shuts down completely the data will not be saved to the hard drive and probably lost forever. Zookeeper has a check-pointing mechanism where the system can most most of data (not including the most recent though). That is a trade-off that was made by the designers of the system to protect the performance of the system rather than providing guarantees that the system has all the data.

2.3.3 Zookeeper Popular Usecases

How is Zookeeper used today?

- It is used as a Naming Services: It is used to identify nodes in a cluster by name. The most similar service could be DNS but for nodes.

- Cluster management: This is how also Kafka is using Zookeeper. It is responsible to add or remove a node from a cluster and also update the status of a node in real time. For example if it is active, pause, resume, cancel state.

- Leader election: Kafka is also using this feature. Zookeeper is responsible for keeping track of the nodes and electing a leader. The leader in Kafka case is a broker. If a broker fails then Zookeeper is responsible for electing the next leader. There are many ways to choose your leader but Zookeeper is using a quorum algorithm to do that.

- It is used for implementing locks and taking care of the synchronization of distributed systems.

2.4 Distributed Messaging Systems

Nowadays, there a great number of tools and frameworks designed to send and receive messages distributed. The majority of these tools are open-source and the development started after the start of the Hadoop project, which was the beginning of the new big data era.

What problem do they solve?

Distributed Message Brokers are typically used to decouple the communication of data with the processing of data. They typically follow the publish-subscribe design pattern. A classic system has a data producer, a data consumer and a storage system(broker).

The producer is writing the data to the storage system, and then asynchronously a consumer could request data from the broker and pull the data that are written to it. The storage system of the broker could be anything from a simple hard drive, SSD, ram, database. It is more common to use a ram first if the requirements and the size of the ram is big enough to save the data and then an SSD because of the high I/O performance. Database is the least used system since it has high latency which against the requirements of a streaming system. The most popular systems that follow the aforementioned pattern are Kafka[1], AMQP[8], ActiveMQ[9], Facebook Logdevice[10], Google Cloud Pub-Sub[11], Amazon Kinesis[12]. All these are alternatives to Apache Kafka and each system has advantages and drawbacks in comparison with Kafka.
For example:

- ActiveMQ : is very fast and it is easy to use with Java based (JVM) applications.

- Amazon Kinesis: is very good at processing large amount of requests in parallel. It is able to write applications that process data in real-time and scale across thousands of nodes. On the other hand, it is a fully managed services which can be used through AWS and there is no open source version to it.

- Facebook's Logdevice has a very rich API and provides improved guarantees compared to Apache Kafka with similar throughput, but there is no open source version to it and there is no really community that is using this tool. The interest in Logdevice is very low there the trend is definitely not going upwards.

2.5 Apache Kafka

Apache Kafka is a distributed, partitioned, replicated commit log service, that provides the functionality of a messaging system. It is used for collecting and delivering high volumes of data with low latency. Apache Kafka was originally developed by LinkedIn, and was subsequently open sourced in 2011. LinkedIn is using Kafka in production for some years already and the system is processing hundreds of GB successfully everyday. At Kafka[1] they compared the system against the most popular messaging systems and the experimental results showed the superiority of Kafka in terms of throughput (messsages/second). In 2012 Kafka became an Apache Top-Level Project. The basic concepts of Kafka are the following:

- Topic: It is a stream of messages of a particular type.
- Partitions: Each topic is divided into multiple partitions. This design choice was made in order to enable a distributed Kafka system to be able to achieve load balancing.
- Producer: A producer is able to publish messages to a particular topic.
- Brokers: Set of servers that pub-messages are saved.
- Consumer: A consumer can subscribe to one or more topics from brokers pull the data from the brokers.

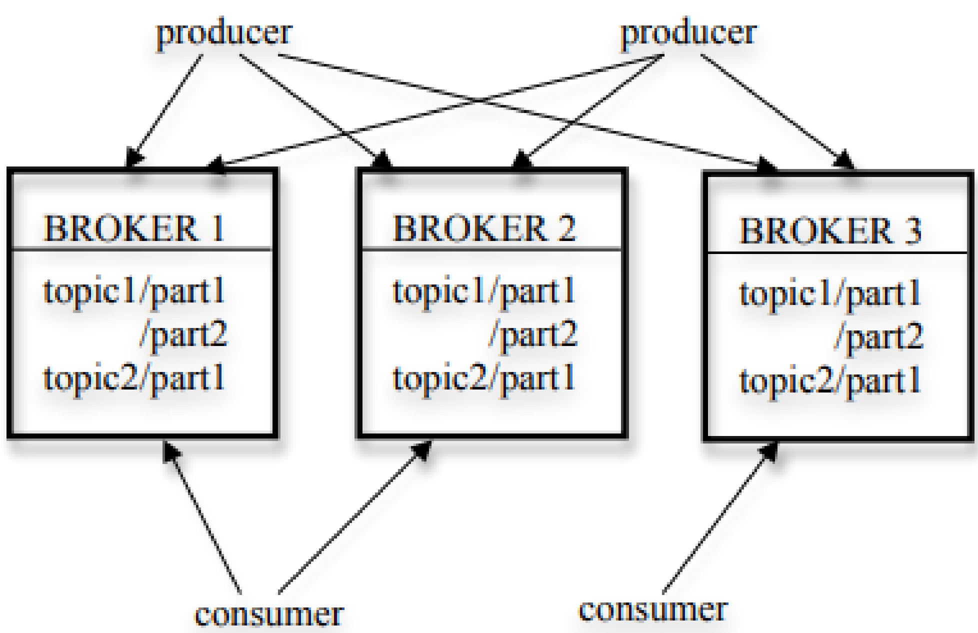

Figure 2.2: Kafka Architecture

2.5.1 Kafka Architecture And Design Principles

Kafka is a distributed messaging system. A Kafka cluster consists of the following: **Brokers**: A typical cluster consists of multiple brokers. To balance load the workload, each topic is divided into multiple partitions. The user is responsible for deciding how many partitions each topic should have. Each broker is storing one or more partitions of each topic.

Producers - Consumers At each topic a system the system can support multiple producers and multiple consumers.

Consumer Group Kafka, has also designed consumer group. A consumer group is a set of consumers where each of them consume data from different partition. That makes data pulling faster and also it avoids the consumers from pulling the same data twice (data duplication).

Kafka has made a few choices to improve the throughput of the system. First, they decided to use segment files to append to new data first, and then flush the segment file to the disk. The data is available for consuming only after Kafka has flushed the data to the disk. Moreover, Kafka decided to reduce the information sent by every message. Unlike typical messaging systems, a message stored in Kafka does not have an explicit message id. Instead, each message is addressed by its logical offset in the log. This avoids the overhead of maintaining auxiliary, seek-intensive random-access index structures that map the message ids to the actual message locations. Note that our message ids are increasing but not consecutive. To compute the id of the next message, we have to add the length of the current message to its id. Below we can see the layout of a Kafka log and the in-memory index is shown in the next figure. Each box shows the offset of a message.

Another choice that was made by Kafka designers in order to keep reducing complexity and overhead at the broker. Kafka has a **Stateless Broker**. That means that Kafka is not keeping track of how much data each consumer has consumed. The drawback at this design is that the broker does not know when it is okay to delete consumed information. If the broker does not delete information the system will reach a point

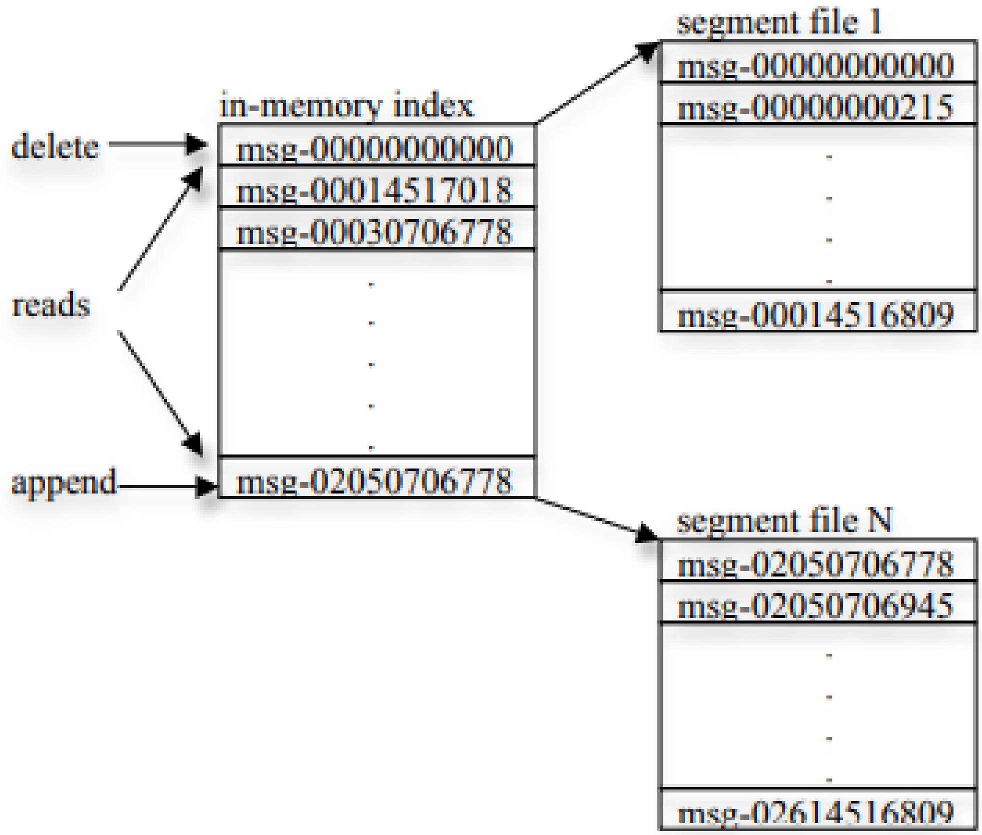

Figure 2.3: Kafka Log

where no more messages will be able to be written to the system. The solution to this problem is to delete messages based on a simple time based policy. The broker will delete information after a time period which is decided by the maintainer of the system. The default time right now is 7 days. At this point it is important to note that the system's performance does not depend on the size of messages that are saved in the broker. Therefore, the performance of the system will note degrade even if we keep data for a longer period.

Last but not least, Kafka decided to decentralize the consumer of the system. That means that Kafka does not follow the popular architectural design pattern in Big Data system "Master - Slave" because adding a master would complicate the system in terms of failures. Failures in master node would take the whole system down and as it will be discussed further more extensively at least one time guarantees would make things more complicated. Kafka used Zookeeper in order to decentralize their architecture. Kafka uses Zookeeper for the following tasks:

- detecting the addition and the removal of brokers and consumers,
- triggering a re-balance process in each consumer when the above events happen,
- maintaining the consumption relationship and keeping track of the consumed offset of each partition.

2.5.2 Delivery Guarantees

Kafka offers at least once guarantees. Exactly-once guarantee does not allow duplicates but on the other hand it makes the communication much slower because it requires two-phase commits. Given Kafka's architecture, if a consumer crashes, the other consumer that will handle the incoming messages may get duplicate messages.

The problem could be solved on the application level. The application should be able to handle the duplicate messages by removing them, or by restarting the application.[1]

Kafka guarantees that messages from a single partition are delivered to a consumer in order. However, there is no guarantee on the ordering of messages coming from different partitions. To avoid log corruption, Kafka stores a CRC for each message in the log. If there is any I/O error on the broker, Kafka runs a recovery process to remove those messages with inconsistent CRCs. Having the CRC at the message level also allows us to check network errors after a message is produced or consumed.

2.5.3 Kafka Fault Tolerance

There is room for improvement in Kafka's fault tolerance architecture. If a broker crashes and there are messages stored on that broker which are not consumed yet, they become unavailable. The problem is that if the storage system which the data are stored becomes permanently damaged all unconsumed messages are lost forever. Kafka development team plans to extend the architecture of the system by automatically replication messages on multiple message brokers. In that way even if a server goes bad then data will not be lost.

2.5.4 Common Kafka Deployment

After explaining the challenges of a messaging system and the Kafka Architecture I am going to explain how a Kafka system should be deployed in order to tackle the challenges discussed above. The following explanation will be a general rule of thumb which could be used as an tutorial for most of the use cases that is required. Every application has different requirements and therefore, if someone knows the need of the application that he is supporting he will be able to optimize/ fine tune Kafka in order to get the maximum performance from the system.

How to solve reliability availability and scalability with Kafka:

Apache Kafka has already implemented, as described above fault tolerance in its architecture. The only way to make sure that the system will be reliable and also available (available means that the system does not go down often, due to maintenance and broker failures) is to have a big number of brokers spawned across many different nodes. As a result, even if a broker fails, then zookeeper will elect a new leader immediately and then the system will never stop operating normally. The broker that failed will restart automatically and if the problem persist then an engineer will have to see the root cause of the issue the resolve it. Moreover, scaling brokers horizontally across multiple nodes will help the system provide high-throughput data transfer that could be sizes from bytes to Terabytes. Adding more machines seamlessly share the load across hosts. In order to share the loads evenly among nodes, a load balance system is required. That could be developed either along with the messaging system or it could also be part of the application that is sending messages to Kafka. Being part of the application, usually increases the overall performance of the system because there is no need for middleman to orchestrate the data between the application and the messaging system. From the consumer's side there is no need to add any load balancer because Kafka has already implemented a consumer group which allows multiple consumers to receive data from the same topic without any extra overhead from the application developer.

Furthermore, multiple Zookeeper instances are required in order to make sure that system will not fail. Brokers are orchestrated by zookeeper so if Zookeeper fails then the whole system will go down. The good thing about ZK is that it is very reliable and it is able to handle many instances without introducing any bottleneck to the system. That being said, 1 instance of zookeeper every a few brokers is sufficient to ensure that the system will be operating normally without issues.

2.6 Distributed Data Streaming Processing Engines

2.7 Apache Spark

2.7.1 Introduction

Apache Spark[13] is a computing platform specifically designed for computer cluster (cluster computing platform) materialized in the programming language Scala . Built to support distributed general purpose applications is generally based on processing large volumes of data with a high degree of efficiency and speed. In a context of the Spark is an extension developer MapReduce model, with the main difference that supports most types of calculations, such as interactive queries (interactive queries) and processing streaming data (streaming data processing). Another difference with the work (jobs) that can be d to Spark in connection with embodiment of MapReduce in other systems is the ability to provide data storage in the memory of each node in the cluster during the execution of the job . This latter property described as caching and is probably one of the most important features introduced by Spark in developing distributed systems and gives advantage in speed over the Hadoop MapReduce even achieved up to 100 times better time performance.

The Spark structure is best described as a single stack of subsystems cooperate and provide each of these services separately. Specifically, the "heart" of the system is the Spark core. The core module is the computational engine system that is responsible for routing, sharing and monitoring applications which are detailed in individual computing units (tasks) and assigned to the cluster nodes. To core is also what provides interfaces for building programmatic elements of the system such as, for example Resilient Distributed Datasets (RDD) that we will analyze later. It is also the basis on which the subsystems based Spark Sql, Spark Streaming, MLLib and GraphX. Briefly mention here that the Spark Sql offers potential for cooperation with the Spark structured data via Sql language and variant that offered by Apache Hive. The Spark Streaming is what gives the possibility to store and stream data in real time. The GraphX is a special library designed to provide support for jobs and algorithms that process graphs. Finally MLlib the library a collection of machine learning algorithms. MLlib is a powerful Machine Learning Library which give competitive advantage to Spark in contrast with other distributed data processing frameworks.

Spark as mentioned above is implemented entirely in the Scala programming language, a multi-language paradigm with strong evidence for Functional and Object-Oriented programming. The scala uses the JVM environment for the execution of its programs, and is designed so that it can import and use libraries of Java. Like the Spark naturally supports application development in any of the Java and Scala language while outside these supports and the Python language. This makes the Spark platform independent (platform independent) while with Python provides great freedom of choice from the users point of view.

2.7.2 Apache Spark's Architecture

Spark architecture which is based on the famous map-reduce paradigm, follows the architectural master / slave model. More specifically in Sparks terminology there are the master and the workers entities. When you start a Spark cluster one master process is running on the node from where the start command is given, while workers processes are executed on slaves nodes. In order to run a Spark application you need compute resources from the cluster, translated into main memory and cpu cores. A supervisor of the available resources of the cluster provides Spark the choice among Standalone, Yarn and Mesos cluster resource managers. The standalone which is one that we will use in this work, and an integrated implementation resource manager that provided from Apache Spark. Master and worker processes belong are controlled from the standalone resource manager which has the supervision of all the processes and the resources a stock of the given cluster. Besides that, we need to maintain continuous communication between workers nodes and master node not only to run the application successfully, but also to be able to identify any problems. Finally these processes set up servers at each node in order to provide useful information of their operation via the http protocol. The main components of a Spark application are the driver the executor and the cluster manager.

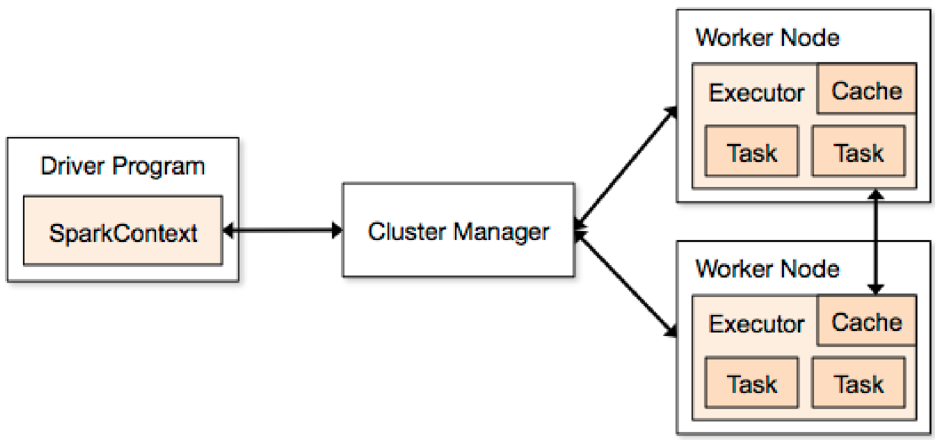

Figure 2.4: Spark Cluster Overview

1. **Executors**

The executors[13] are Java processes that start the workers in the cluster nodes. The executor initiated from the master node at the beginning of each job and the duration life is as much and the lifetime of the job. The executors are those who take part to the calculation needed for the job. This is done by assigning independent calculation units (tasks) in each executor. Specifically sending each executor is to commit the necessary resources, memory and CPU cores, and perform specific operations laid down by the code of the job on a track (partition) of data processed by the job. The results of their work vary depending on the type of task. These either communicated back to the driver to monitor the implementation process, as we shall see, either stored in the cache or disk as intermediate results that will be consumed by another operation in a future task. This storage is undertaken by a service called BlockManager and cooperates with the executor and the driver. Finally it is important to emphasize the fault tolerance of

Sparks Architecture. Each task is an independent compute unit, which means that if for any reason a task fails to complete, that does not affect the ongoing execution of the job only because another identical task will be be re-initiated.

2. Driver

Driver is the process which controls the main method of the client application. In this there is the object SparkContext represents and encapsulates all selected settings and parameters for a particular job. The two main obligations driver process can be summarized as follows: Driver process is responsible for breaking the application into smaller tasks/pieces. When the application starts running , the drivers responsibility is to convert the code into tasks and delegate them to executors that are available to the workers of the cluster.

In the beginning, the driver draws a logical plan of job , which has the form of directed acyclic graph, (Directed Acyclic Graph - DAG) of the procedures to be executed. After that, the logical execution plan needs to be translated into physical execution plan. At this stage undertaken various optimizations in the way of execution that can be made by the driver. Finally translate the logical implementation plan in natural execution plan which consist of multiple tasks. The second step is to schedule the tasks to the executors. The routing of tasks to the executors take place right after the output from the physical execution plan driver for the job.

In this phase each executor has made his presence known to the driver and expects it to be assigned a task to start computing. The driver makes smart choices assigning as much as possible task nodes that are stored locally and the data will need to edit this task. These may concern either data stored in the local node disk and accessed for the first time or even data stored in the cache node by earlier task performed there. In either of the above two cases, the driver will always prefer the particular node that meets these criteria. This of course is not always possible because it only has to think about the case where no such nodes in our cluster, but at this moment is busy performing another task. However, it is more preferable to select such a node in connection with any other, the spark and provide relevant regulation specifying the number of seconds that can expect the driver to be released as a node eventually to delegate the task.

2.7.3 Resilient Distributed Datasets (RDD)

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes. Formally, an RDD is a read-only, partitioned collection of records. RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.

There are two ways to create RDDs parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format. Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations. Let us first discuss how MapReduce operations take place and why they are not so efficient.

Data sharing is slow in MapReduce due to replication, serialization, and disk IO. Most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations. Recognizing this problem, researchers developed a specialized framework called Apache Spark. The key idea of spark is Resilient Distributed Datasets (RDD); it supports in-memory processing computation. This means, it stores the state of memory as an object across the jobs and the object is shareable between those jobs. Data sharing in memory is 10 to 100 times faster than network and Disk.

2.7.4 Caching

As we have seen so far in Apache Spark[13] introduces the field of distributed computing machines some very interesting facts, such as RDD we analyzed in the previous section. What I generally do it very attractive for data analysis applications (data analytics jobs) is the feature that allows stored during execution of a job intermediate results (which are themselves in turn RDD) in the main memory of the nodes the cluster, which is called caching. Traditional computing models such as MapReduce, which follow a similar philosophy of execution can only preserve data in the main memory within which they are processed by a given task at a given time. If the same data are needed from a next task to be executed in the same node then those provided by the disc. Obviously it creates a weak point (bottleneck) in carrying out the task as the data access speed of the main memory in comparison to the disk are hyperproliferative. Very significant benefits from caching reap the jobs those using iterative algorithms (iterative algorithms), processing at each iteration the dataset which are powered. The k-means belonging to this class of algorithms and appropriate in this section to look in a little more detail the concept of caching and how it is implemented in Spark. As we have already seen the executors are those java processes running on each node of the cluster and make the core of calculations each job. Since it is java processes running in a Java Virtual Machine (JVM) that provides the memory piece known as heap, which is the available process interface (workspace). To heap is divided into partial memory pieces which serve to store different kinds of data related to the process. Spark has a default size of 512MB for each executor which can be tuned using spark.executor.memory configuration, based on our needs. Spark expresses its memory size based on the total workspace of each JVM[14] executor. We have the following regions:

- **Safe Region:** This region accounts for 90% of the total heap and so called precisely because it sets an upper limit which can engage the heap to avoid low-memory exceptions. (Out of Memory Error - OOM)

- **Shuffle Region:** Occupies 20% of the safe as shown in the figure and is designed to store data which will shuffle. The procedure takes place when shuffle data produces a task that must be consumed by another node that runs another task in the cluster. Usually these data are required to be classified and so this part fulfills the memory requirements for such classification.

- **Storage Region:** Accounts for 60% of the safe region is what serves the storage partitions an RDD at a node. From here they can access the data much faster than any task.

- **Unroll Region:** Spark also provides the ability to store data in serialized form in memory or on disk. This data format required by distributed systems such as Spark in order to move data over the network. In serialized form data but can not be used immediately, but first a de-serialization process has to take place. These operations fills the unroll region which is 20% of the storage region.

2.7.5 Spark Streaming

Spark Streaming extends Spark's API to stream processing which let you develop streaming jobs the same way that you would write batch jobs. That makes the transition from batch to stream pretty easy and it also enables code reusability because a developer can develop only one software for two different purposes.

Fault Tolerance: Furthermore, Spark Streaming is also Fault Tolerant. The systems requires exactly one semantics from every state. In case of a failure the system is able to recover from it automatically and execute again the faulty state. This is extremely important for close to real-time processing where you need to detect and fix the error immediately.

Deployment Options: Spark Streaming[15] is able to read data from many different data sources such us HDFS[16], Flume[17], Kafka[1], Twitter[18] and ZeroMQ[19]. Furthermore, a developer can also define his/her own custom data sources. Therefore, integration with Apache Kafka which is used at the current project is extremely easy and well supported by the community.

2.8 MapReduce

MapReduce[20] was presented by Google in 2004. It is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The runtime system takes care of the details of partitioning the input data, scheduling the programs execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: Map and Reduce. Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function.

map (k1,v1) list(k2,v2)

The Reduce function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation. The intermediate values are supplied to the users reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

reduce (k2,list(v2)) list(v2)

2.9 Spark Streaming - Architecture

In general, modern distributed stream processors take three steps in order to do streaming.

- Step A: Receive data.

- Step B: Process data.

- Step C: Emit result.

Spark Streaming[21] which is mini-batch streaming does not process the data immediately after receiving them. Spark first splits the data into batches and convert them to RDDS. Spark then every few seconds, which is user defined, (it is called streaming window) takes the collected RDD and process it. That helps spark to process many data in parallel and also re-use the existing spark software for batch processing. Therefore, spark can perform better load-balancing and also recover fast for failure in nodes. Moreover, the main reason that Spark has very high throughput compared to direct streaming systems is that Spark is processing data in micro-batches. The drawback of this method is that latency is higher compared to direct streaming frameworks.

The input data stream is represented at spark by a high level abstraction called discretized stream or DStream[21]. Each RDD contains data from the same time interval. Any operation applied on DStreams are converted to operations applied on RDDs as shown to the following image.

Developers perform operations on the DStreams, which are represented as I said earlier with RDDs. Re-using RDDs does not benefit only the developers of spark, which let them re use the same code they developed for batch processing but also the application developers which can do both batch and stream processing with very small changes in their code base.

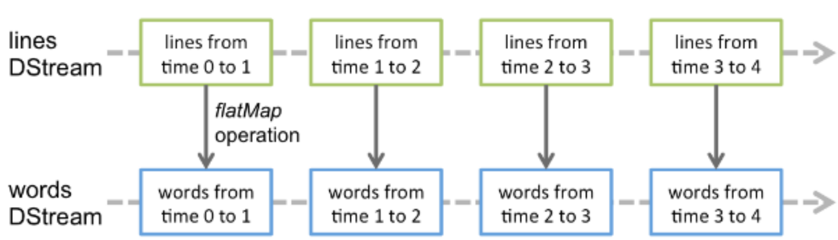

Figure 2.5: Operations on DStreams are converted to RDDs at each window time interval

Spark Streaming can ingest data from many different sources. The most popular sources are: Kafka, Flume, HDFS/S3, Kinesis[12] and Twitter. Any TCP socket can be used to send data to Spark. The benefit of using one of the aforementioned messaging systems is that people have already implemented connectors with Spark which makes the integration super easy. After finishing the data processing Spark can emit data to file-systems, HDFS, databases dashboards.

Figure 2.6: Spark Streaming input and output to various sources

2.9.1 Spark Structured Streaming

Structured Streaming[22] is Apache's Spark for continuous low-latency Streaming. Spark developed a new High-level API which is now part of the Spark core engine. Structured Streaming was first introduced in version 2.2.0. Looking at the programming guide of structured streaming one could easily understand that data processing is mainly done using the Spark SQL engine. The idea behind this choice was made in order to enable users to still process streams with classic database processing operations (like filter, group, aggregate) in a continuous manner but the information would be processed by Spark Engine using micro-batch style operation. Spark really improved Structured Streaming in 2.3.0 version where they achieved end-to-end latency less than 1 ms. Streaming could be considered low latency and it could compete with other low-latency Continuous Streaming Engines like Twitter's Storm[23], Heron[24] Flink[25] etc. At the following figure we can see Spark's architecture of Structured Streaming with a quick example.

Spark is appending data to a table continuously. Every millisecond new data is appended to the table and the user can immediately process the data. Moreover, Spark extended its API by including DataFrame and Datasets API. That will make developing applications for Data Scientist much easier than it used to be with Spark.

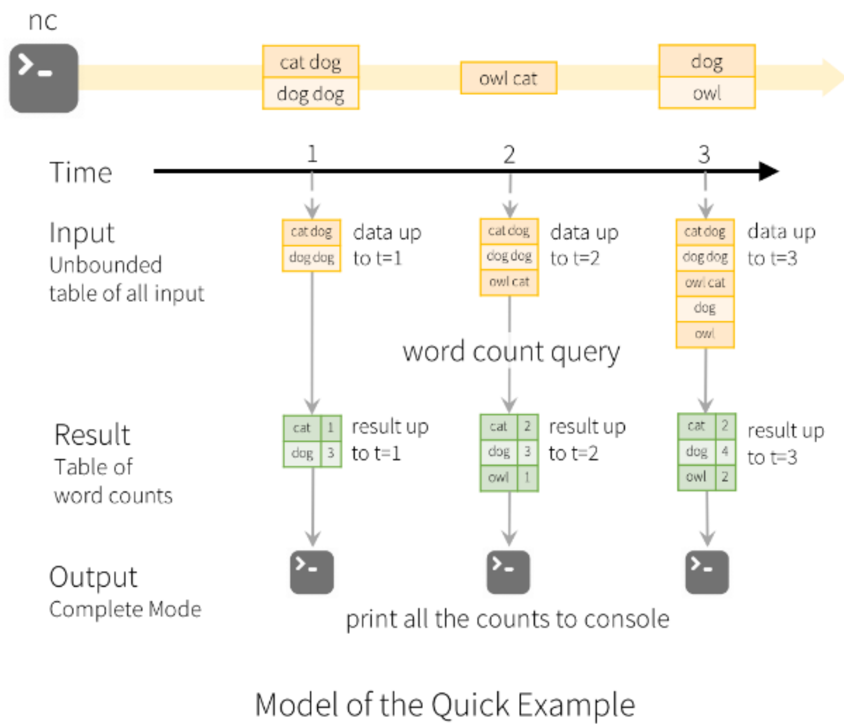

Figure 2.7: Spark Structure Streaming Quick Example

2.10 Integration Of Spark Streaming With Kafka Broker

Since at this work we are using Spark Streaming with Kafka, I am going to highlight the different ways of integration only with Kafka. There are two main Approaches in Integrating Spark Streaming with Apache Kafka:

- Approach 1: Receiver-based Approach

- Approach 2: Direct Approach (No Receivers are used in this approach)

Each method has different advantages and drawbacks. What is common about these methods is that the Spark stores the data in both cases in Spark executors.

2.10.1 Receiver-Based Approach

This approach is using a Receiver to get data. The drawback about using a receiver is hte fact that under default configuration system might data under failures. Spark in future versions added the capability to write-ahead logs in Spark Streaming which will ensure that no data will be lost. How does Spark do that? It saves in a synchronous manner all the logs that are received from Kafka on a distributed file-system, and data will be retrieved on failure.

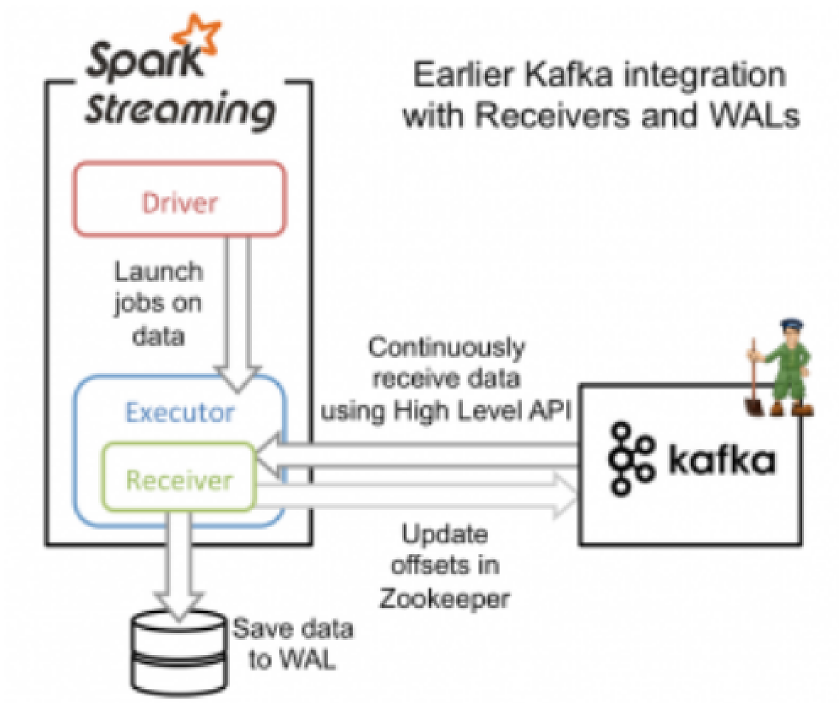

Figure 2.8: Spark Receiver based architecture

At this point it is important to highlight some points about Kafka and Spark Streaming:

- There is no correlation between Kafka Partitions and RDD Partitions generated in Spark Streaming. That means that by increasing the number of Kafka partitions will increase the number of threads where data are consumed but it will not increase the level of parallelism that Spark will use to process the data.

- For parallel Receiving of data one could send data over different topics, because each receiver is receiving data from one 1 specific topic.

2.10.2 Direct Approach (No Receivers)

The Direct Approach was developed after the receiver-based approach. It was introduced in Spark 1.3 and its main goal was to to ensure better end-to-end guarantees. At this approach Spark is not waiting to receive data from but it is polling periodically data by querying the latest offsets from Kafka. This approach is considered much more advanced and it will be the one that I am using at the applications and experiments that I developed for this thesis. The main metrics that the receiver-less approach is doing better are the following:

- a) Automatic Parallelism: This method will automatically query the number of Kafka partitions and create equal number of spark RDD partitions in order to do one-by-one mapping. This method is not always the most efficient, but this mapping makes it much easier for the developer to understand what is happening within the system and it will be easier for him to improve and fine-tune the system.

- b) Efficient Fault Tolerance: Direct approach does not need write ahead logs but spark is able to query the lost data directly from Kafka. The main advantage of this method is that the Spark does not have to use the write-ahead log method which saves data to the disk and save both cpu cycles that write data to the disk and disk size.

- c) Exactly-once semantics: The first approach uses Kafkas high-level API to store consumed offsets in Zookeeper. This is traditionally the way to consume data from Kafka. While this approach (in combination with-write-ahead logs) can ensure zero data loss (i.e. at-least once semantics), there is a small chance some records may get consumed twice under some failures. This occurs because of inconsistencies between data reliably received by Spark Streaming and offsets tracked by Zookeeper. Hence, in this second approach, we use simple Kafka API that does not use Zookeeper.

Offsets are tracked by Spark Streaming within its checkpoints. This eliminates inconsistencies between Spark Streaming and Zookeeper/Kafka, and so each record is received by Spark Streaming effectively exactly once despite failures. In order to achieve exactly-once semantics for output of your results, your output operation that saves the data to an external data store must be either idempotent, or an atomic transaction that saves results and offsets. (see Semantics of output operations in the main programming guide for further information).

Drawback:

Direct Approach does not update offsets in Zookeeper as most of the traditional system. As a result, all the Kafka monitoring tools that are developed based on that logic are useless. That is not very important though, because the developers could manually update the zookeeper offsets and get reliable data from the tools and also tools are being developed continuously therefore, soon there will many tools that does this process automatically.

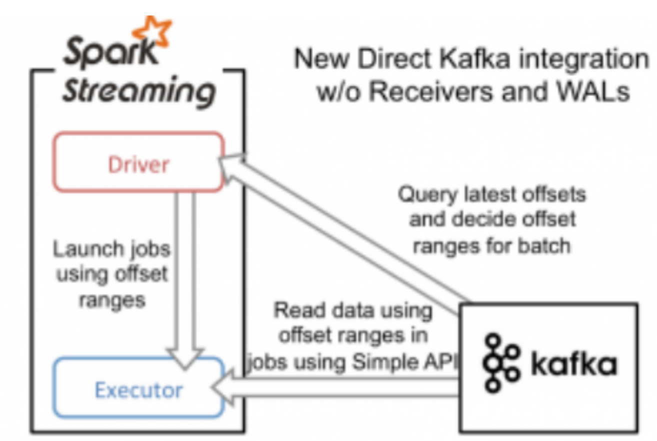

Figure 2.9: Spark Direct Approach (Receiver-less)based architecture

2.10.3 Receivers

The above information for receivers refer specifically to the Kafka integration but the idea is the same for all receiver based systems. For example: a receiver will never ensure data fault tolerance unless another subsystem will take care for that.

Chapter 3

Applications

3.1 Applications On Connected Cars

What applications could benefit from the architecture that I developed at the current thesis?

Could the cloud provide information to help the car to break and prevent an accident? What is the requirement for that?

The brake reaction time is the amount of time that elapses between the recognition of an object or hazard in the roadway and the application of the breaks from the driver. Based on the research that was performed at the Idaho university[26] the length of the brake reaction time is different from driver to driver. They clustered the drivers into two different categories:

- The alert driver: which is able to react in less than a second.
- The average driver: which usually need from 1 to 3.5 seconds to react.

There are many reasons that could make the reaction time vary:

- Driver characteristics: Such as the level of fatigue, experience, age, alcohol consumption.
- Environmental conditions: Such as weather and visibility, lightning conditions etc.

In general, 90% of the population is able to react in less than 2.5 seconds. The system that I designed will be need at least a second to make a decision. Therefore, more than 10% of the population will be able to benefit from such a service but it can only be auxiliary to the driver. A system that can perform computations in sub-second manner will be able to solve this problem reliably.

Problems that the current architecture will be able to support:

- Providing real time information to the users about road conditions. Hazard could be identified by one car(example: potholes) and will be shared to the cloud, so the pathplanner will be able to take that into consideration and other cars might avoid to pass from the same spot. For example: If one car falls in a pothole the others could drive there very slow with caution or take an alternative router to avoid it.

- Cars that have many sensors like cameras, could use a cloud system to update their machine learning models. For example: A trained model will be able to use the input from the camera sensors, make decisions and then compare the decisions of the model with driver's decisions. As a result, the car will learn the habits of the driver and try to drive like him. That means that the system will help develop a personalized self driving software.

  Let's remember a problem of Tesla's self-driving car that lead to a fatal accident. the car did not manage to see that it was driving towards another object which happened to be a car with a human in it. The problem was that the light exposure was so high that the vision algorithm failed to detect it and Tesla crushed on it. The result was that both drivers died. It would be very easy for the platform that I am proposing to detect the accident, by using simple information, like the geolocation and help avoid the accident.

- Another problem that could be solved using this architecture is the traffic congestion. Cars could share their location and their speed and then the system would gather the information and use some techniques (the most accurate techniques are machine learning models today) to predict in real time the traffic. Again, this will help cars make better decisions for their routes.

The aforementioned applications have the following requirements:

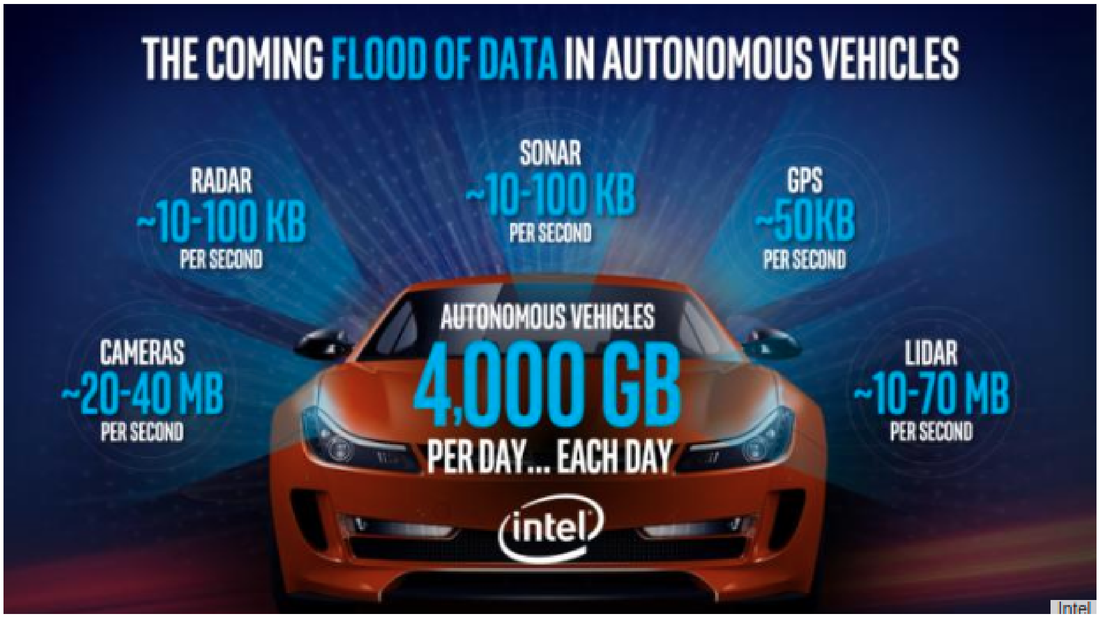

Figure 3.1: Data from autonomous car

- The cars need an internet connection and a reliable messaging system to send and receive data from the cloud.

- a cloud system which is going to gather all the data from the cars, process the data in real time and send results back to the cars if asked.

- Make sure that the data will be transferred safely to the cloud. This is very important because if a hacker takes control of this system it will be able to influence the decisions of the cars on the road and create fatal accidents.

- The system should be able to handle large amount of data. Based on Intel[27], just one autonomous car will produce more than 4TB of data/day. Therefore, large scale computing and processing is a really important requirement.

Chapter 4

System Overview

The system that I propose to solve the problem of communication between Internet Connected cars will be based on two components:

- Apache Kafka: Which is going to be used for publishing and consuming messages from the cars to the cloud and vice-versa.

- Apache Spark: Spark will be used for processing the consumed messages from Kafka in real-time.

The goal of these architecture is to achieve the following:

- Make sure that data will be sent and received in a fault tolerant way. (Kafka can guarantee that with the correct setup.)

- Make sure that the system will have adequate throughput to send are receive messages from many cars in the same area. This is something that Kafka always promises that it does. Later, I will do experiments with realistic applications that will show that this is feasible using Kafka.

- Make sure that messages will be processed in realtime in order to pass information fast back to the cars and help them make the right decisions.

- The system will be able to process the data in a fault tolerant way, to ensure that it will not lose important data from the cars. This is something that Spark also guarantees.

Spark and Kafka seems to cover the requirements of a broad range of internet connected cars applications and they will be able to serve them well. Moreover, as I mentioned earlier there are already implemented connectors between Spark and Kafka and therefore these two tools are already tested in production and are ready to use. Bellow we can see in the Diagram the architecture that I built for the current thesis.

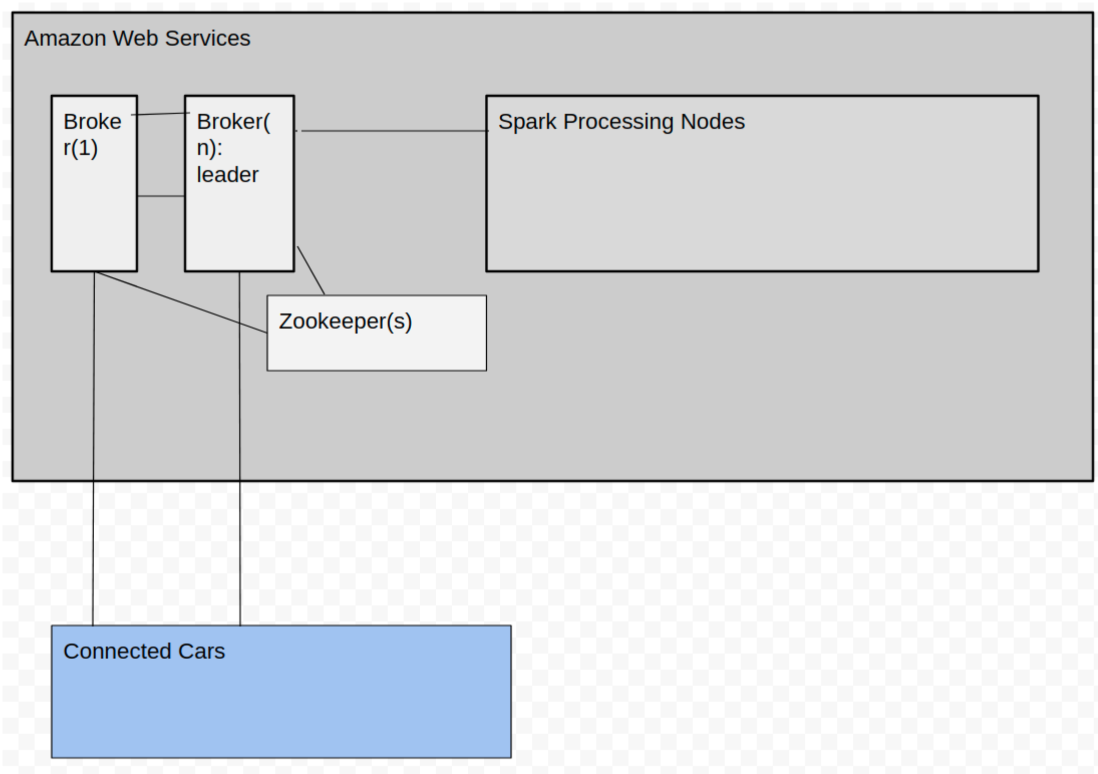

Figure 4.1: System Architecture

Amazon Web services hosting all the software Stack, which is Kafka, Zookeeper and Spark. Ideally, all the hardware nodes are positioned really close to each other. That will help reduce the latency between the communication between the different nodes, but especially between Kafka, and Spark which latency is really important. Latency between Zookeeper and Kafka could be a little higher without affecting the performance of the whole pipeline.

Kafka will be responsible for communicating with the "rest of the world". Connected cars will be sending and receiving messages to Kafka. Kafka will talking to the Spark Nodes whenever is required.

4.1 Monitoring Tools

Monitoring A Spark-Kafka Zookeeper architecture is not a very easy job. The pipeline has a very big number of different parameters that need monitoring. The most difficult part is the monitoring of the end-to-end architecture because everything should be working in in perfect harmony. That means that Kafka should be producing data in order to monitor the performance of the Spark Nodes.

To help mitigate this problem, Amazon Web Services have developed their own monitoring tool which helps measuring the performance of the pipeline. The name of this tools is Amazon Cloud Watch [28].
Amazon MSK is offering three different monitoring levels:

- The default monitoring level, which provides information for the whole system.
- The PER-BROKER monitoring level, which is able to provide different statistics for every broker of the system.
- The PER-TOPIC-PER-BROKER monitoring level, which is able to provide all the above metrics, but also it is able to provide isolated statistics per topic.

The default monitoring system is free of charge at AWS, but the other two levels come with some extra cost. Usually, using the default monitoring setup is enough to characterize a system. It is also the most popular choice for the following reasons:

- The cost for using the extra service is very high.
- When characterizing the performance of the whole pipeline, I do not need to understand the performance of each broker, but I indent to understand the performance of all the brokers today. In the future, if the brokers are not able to scale (based on the results of strong and weak scaling experiments), it would make sense to measure the performance of each broker separately.
- Similar the above is true for the per topic performance. At first I need to complete the initial experiments which are going to show the performance of the whole

system because trying to analyze the details of Kafka.

- Usually it is enough to measure the performance of the whole pipeline.
- With the right set of experiments, usually it is enough to deduce information that you would have to pay a premium to acquire.

For the purpose of this thesis I developed my own monitoring tools that measure similar metrics as I would do with amazon. The benefit of developing you own monitoring tools is that it comes free of charge and also you can use them in different cloud providers. In case you want to test in a localhost system, or you want to switch to another cloud provider you don't have to learn new monitoring tools.

Chapter 5

Experimental Setup And Evaluation

5.1 Overview

In this chapter we present how we designed the experiments to study the performance of the three different parallel techniques that we are studying. We start the chapter describing the whole experimental procedure that was followed to extract the data that we relied for the stages of analysis and modeling.

In order to understand the performance of the system I have designed the following experiments:

5.1.1 Kafka Producer Experiments:

1st set of experiments

The first experiment that I conducted was to understand how many connected cars a Kafka broker would be able to sustain. To do that I developed a simple application which what it does is to send a message with the x,y coordination of a pothole that detected on the street. The message size for this particular experiment is pretty small.

I measured the total time to send a message to the broker, and I increased the number of different cars that were sending this message. The results from this experiment are discussed after the following plot:

In the plot we have the following observations: The production rate does not change if we increase the number of partitions. Higher number of partitions is expected to increase the parallelism on the consumer side. Therefore it is positive the fact that it does not reduce the producer rate. The throughput rate is not very high. It is 120 messages/sec per producer. It is sufficient to send the data to the system because each

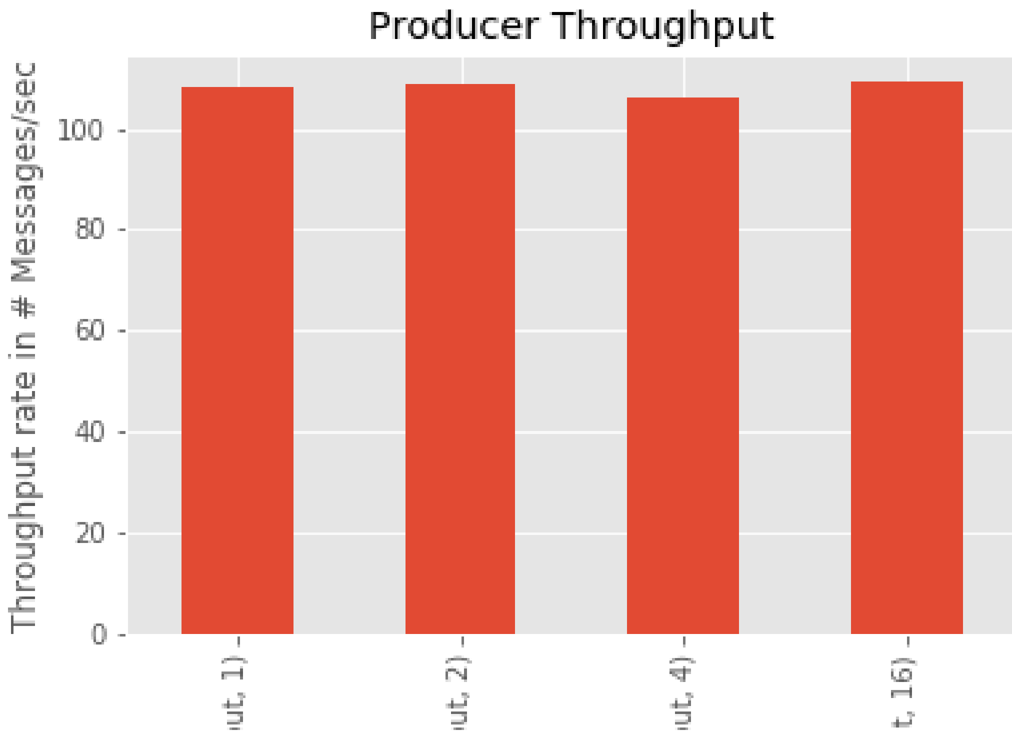

Figure 5.1: Producer Throughput of (x,y) coordinates. Data size is: 61bytes/message

message is 60 bytes. Therefore the system will be sending 5 messages per second. A car that is moving at a speed of 60 miles per hour is traveling 28meters/second. Therefore the system will be able to update its position 5 times every meter of motion.

Moreover, we should take into account the network speed. A good internet connection has a latency of 19 ms and a speed higher than 6MB/s. A small overhead connection is added to the system even if the connection is fast and stable. As a result, the system cannot be used reliably for vital car decisions that need to be addressed in sub second time.

2st set of experiments

In another use-case a car would have to send a bigger message, for example a camera image from the street. Would Kafka this time with a bigger message size be able to sustain the throughput rate? At this set of experiments I did exactly the same setup with the above, but the only difference is that I changed the message from text to image. The size of the image is 335KB. The image that I used for this experiment is the following:

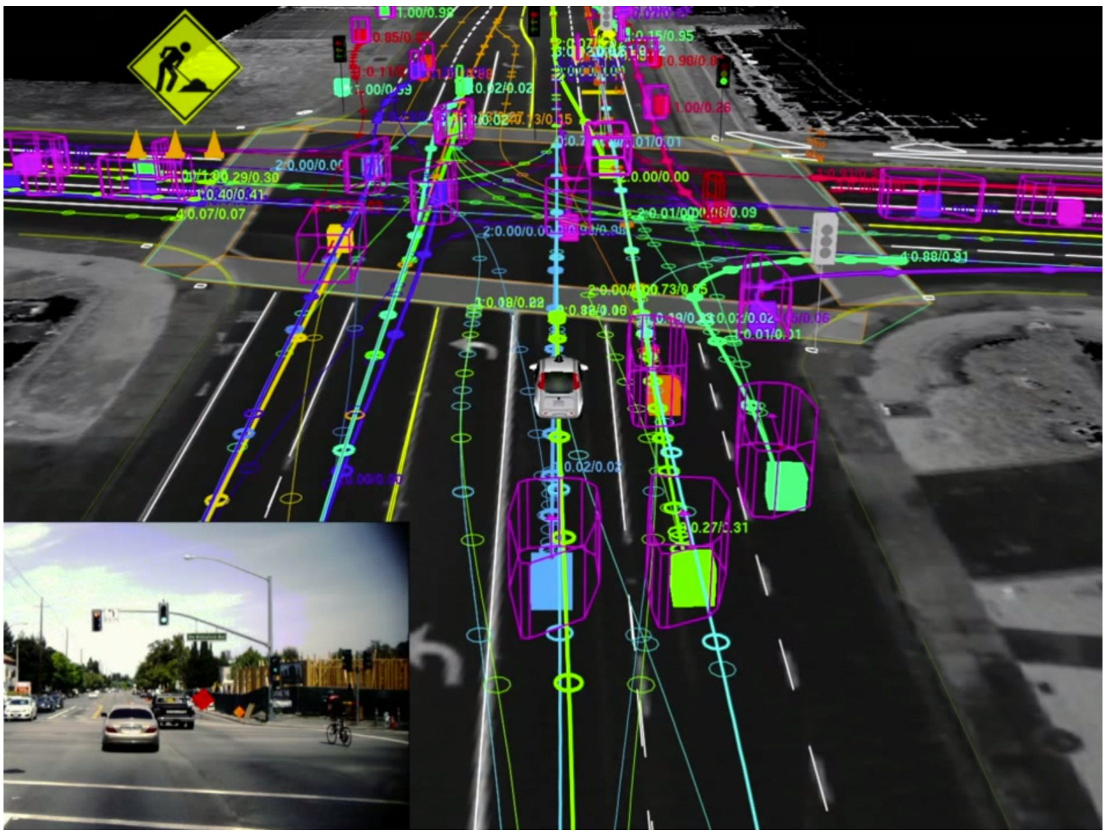

Figure 5.2: Image from the camera of the Car

At the following graph there are some notable observations:

Sending an image, which is a much larger message still is not affected by the number of partitions. We observe the same behavior with the above experiments. The number of images this number is significantly lower. The system is able to send 60images/second which is equivalent to 2MB/second. Compared to the above experiments we can see

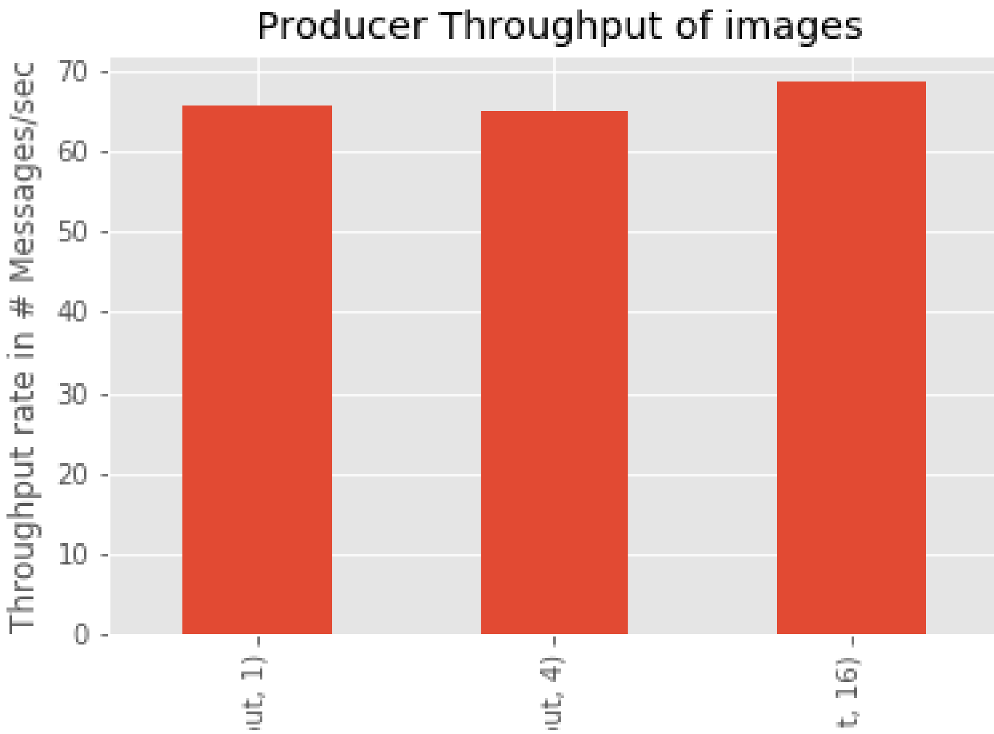

Figure 5.3: Producer Throughput of images. Data size is: 33Kbytes/message)'

that the throughput rate is much higher. 2MB/second is not enough to send all the data back to the cloud that Intel is expecting to be produced by a camera. As a reminder, each car is expected to produce around (20-40) MB/s. Therefore a single producer is not enough to sustain the future traffic.

Another option is to try simulations producer processes. From the experiments that I ran I found that the system can speedup up x16 without any issue by running 16 producer processes. If the car is able to utilize the method of parallel producers it will be able to sustain the required throughput.

3rd set of experiments

At this experiment I am measuring the time that it takes Kafka to consume the string messages that the producer sent at the 1st experiment. From the plot we can make the following observations:

- The consumer does not scale at this experiment. I do not have enough data points to understand the root cause but it seems that if the message size is very small, Kafka has a hard time scaling across consumers.
- The consumer throughput is higher than the producer's throughput. Therefore, consumer will never be a bottleneck even if it does not scale across consumers.

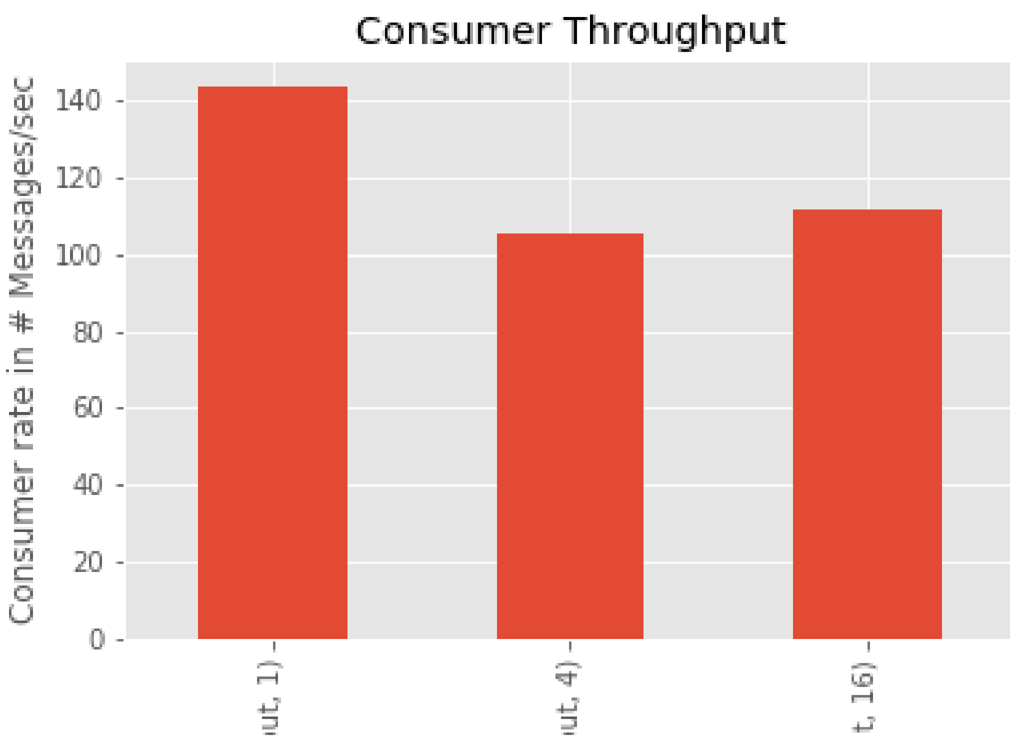

Figure 5.4: Consumer Throughput of (x,y) coordinates. Data size is: 61bytes/message

4th set of experiments

At the following graph there are some notable observations:

- Increasing the number of partitions seems to improve the performance of the system. The system scales linearly from 1 to 4 partitions. Performance is still improving up to 16 partitions but the rate is sub-linear. The different at this experiment is the message size, which is 33KB.
- Consumer with 1 partition has similar performance with the producer. Consumer's throughput is much higher though if it has higher number of partitions.

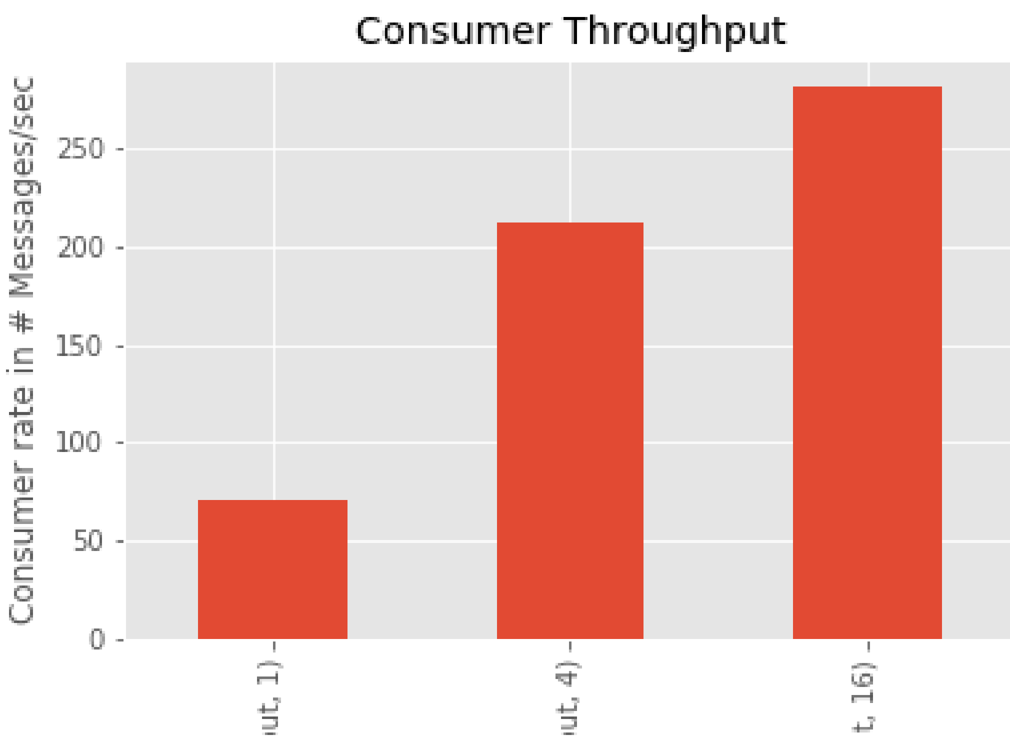

Figure 5.5: Consumer Throughput of images. Data size is: 33Kbytes/message)'

5.1.2 Kafka Consumer Experiments

How fast can the cars pull data from the Kafka?

In order to answer this question I am performing similar experiments to the two that I discussed above but this time cars should pull data from Kafka and not send.

3rd set of experiments The setup that I followed is the following:
I pull from data x,y coordinates and measured the total time to receive one message. I kept the message size constant and I varied the number of cars pulling data from the cloud.

4th set of experiments I performed similar experiments but this time I changed the size of the message that I was pulling from the cloud to the image that I uploaded above. I varied the number of cars that were pulling data from the system.

5.1.3 Kafka Spark Streaming Experiments As A Pipeline

Now that we have a basic understanding about the performance of the first component of the system, I am going to perform end to end experiments to understand the performance of the whole system as one. To do that I developed the following application.
I developed **Traffic Congestion** application. The application does the following things:

- Cars send using Kafka and speed and their location.
- Kafka is collecting the data.
- Spark is consuming data in a window time of one minute defined by the user and processes the data.
- Every window time spark is updating the traffic congestion status of each area.
- The application classifies traffic as low - medium - high based on user defined rules.

Experiment Setup At this experiment I setup 20 parallel producers(cars) that would generate string messages with their location and speed (I ran tests up to 60 producers).
The average rate of each producer is (according to experiment No. 1) 120 messages/second. Therefore the system will be consuming 20*120 2500 messages/second.

Spark Configurations:
Spark Cores : 12

Executors: 1

Streaming Window: 1 second

Application: Traffic Congestion

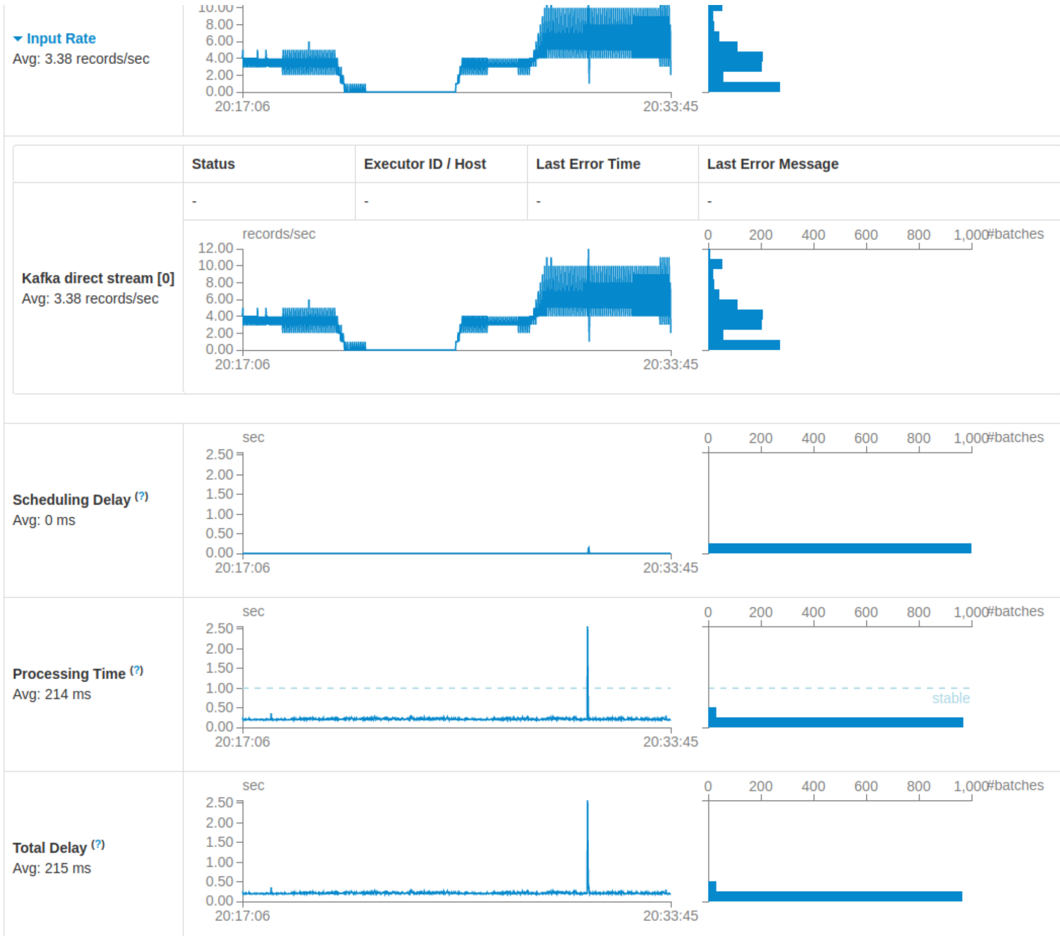

Figure 5.6: 'Spark Streaming Statistics'

From the figure above we have the following observations:

Spark has no problem processing the workload in only 214ms, which is a lot less than the 1 second window that I defined for the application.

Increasing the number of records (by increasing the number of producers) did not affect the processing time. On the other hand, this is very application specific. If the application is computationally expensive it could delay the whole system.

Even when I increased the number of producers the scheduling delay didn't change. It stayed zero ms. The average scheduling delay is the time that the incoming records should wait before being accepted by the engine for computation. The Delay could be affected by 2 main causes: 1) If the number of records is very high and it takes time to convert them to RDDS. 2) if the previous batch did not finish on (window) time, because processing takes longer and as a result the system will delay the next batch. From the figure, we can also observe that during the 10 minutes of execution, there was only one time there was a scheduling delay. That was probably caused by high number of error messages. The system recomputed some data and as a result there is a spike in Processing Time too.

To conclude, the system seems to be able to easily handle all 60 producers using only 1 node with 12 cores.

Chapter 6

Conclusions And Future Work

6.1 Conclusion

The aforementioned system will be able to support real-time streaming applications that produce a very large amount of data. This capability is very important because the amount of data that we produce every day are growing exponentially. As Intel[27] discusses in their report, one self driving car alone will be producing more than 400000 GB of data within a day. We should manage to filter out and process the data in order to extract the useful information because it is already a very profitable business which is going to keep growing over the next few years, we saw that a streaming system has so many distinct variables that could affect the performance of the system. Optimizing one component will then move the bottleneck on another part of the system. It is important, a streaming system to analyzed as one pipeline, in order to make sure that the performance of the whole pipeline is increasing. Moreover, for the same reason, due to the very big number of parameters of the system, it is not going to be easy to characterize it. Therefore, it would be useful if researchers could identify the most important parameters that affect the performance of a system and publicly/openly/officially document them, so that the research community will be able to use it as base guidelines in optimizing their system and extending the understanding of such a software pipeline.

This work is only a starting point in data management of real time streaming systems. There is a number of suggestions that could be analyzed and implemented in the future as an extension to this work. For example it would be interesting to:

- Develop scenarios where the system will have multiple cloud consumers of data.

A good use case that would serve that is the following:

Imagine a car that would be able to send data in real-time to the tire manufacturer. The car automatically can gather data from the sensor and send them back to the tire company. Their cloud system will be able to analyze the data from their tires and provide useful insights to their customers. The car should have the capability to connect to multiple servers, each server will provide data for different part of the car.

- Analyze the capabilities of different streaming processing system. I think Apache Storm[23] would be a good system to analyze because it has the ability to perform continuous streaming is sub-second manner which is something that Spark[15] is not able to do.

- Have another device other than the filesystem taking over the read and write job of the broker. The drawback of the filesystem is that it has limited I/O performance. As a result, it could easily become the bottleneck of a system especially if the rate of data production is growing.

- Explore the performance of the system will different machine learning algorithms. That would be extremely useful since nowadays the popularity of Machine Learning and Deep Learning algorithms is constantly increasing. Moreover, there are multiple ML algorithms that are used especially in the self driving car industry[29].

- Run experiments with much larger hardware systems. Due to financial limitation that would be something difficult to be examined in the current work.

- Explore the usage of GPUs to process the data in real-time. There is already a tensorflow[30] plugin[31] that could be integrated to Spark, and it could serve as an extension to the current system. GPUS are extremely useful in matrix computations[32] and therefore in ML applications.

Bibliography

[1] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. 2011.

[2] Shreyas Chaudhari. Apache kafka in action - better programming - medium. `https://medium.com/better-programming/apache-kafka-in-action-6e19836c9d7b`, 17 2019.

[3] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.

[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. 2003.

[5] David Chappell. Introducing the azure services platform. *White paper, Oct*, 1364 (11), 2008.

[6] Georgios Chantzialexiou, Andre Luckow, and Shantenu Jha. Pilot-streaming: A stream processing framework for high-performance computing. In *2018 IEEE 14th International Conference on e-Science (e-Science)*, pages 177–188. IEEE, 2018.

[7] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8. Boston, MA, USA, 2010.

[8] Steve Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, (6):87–89, 2006.

[9] Activemq. `https://activemq.apache.org/`.

[10] Mark Marchukov. Logdevice: a distributed data store for logs. https://code.facebook.com/posts/357056558062811/logdevice-a-distributed-data-store-for-logs/, 2017.

[11] Google. Google pub/sub. https://cloud.google.com/pubsub/, 2017.

[12] Amazon. Amazon kinesis. https://aws.amazon.com/kinesis/, 2017.

[13] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[14] Kwangman Ko. Java virtual machine. 1997.

[15] Spark streaming + kafka integration guide (kafka broker version 0.8.2.1 or higher) - spark 2.4.4 documentation. https://spark.apache.org/docs/latest/streaming-kafka-0-8-integration.html.

[16] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. The hadoop distributed file system. In *MSST*, volume 10, pages 1–10, 2010.

[17] Steve Hoffman. *Apache flume: Distributed log collection for hadoop*. Packt Publishing Ltd, 2015.

[18] Twitter. it's what's happening. https://twitter.com/?lang=en.

[19] Martin Sústrik. Zeromq. *Introduction Amy Brown and Greg Wilson*, 2015.

[20] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[21] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 423–438. ACM, 2013.

[22] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[23] M Tim Jones. Process real-time big data with twitter storm. *IBM Technical Library*, 2013.

[24] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2742788. URL `http://doi.acm.org/10.1145/2723372.2742788`.

[25] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[26] Brake reaction time. `https://www.webpages.uidaho.edu/niatt_labmanual/chapters/geometricdesign/theoryandconcepts/BrakeReactionTime.htm`.

[27] One autonomous car will use 4,000 gb of data per day — network world. `https://www.networkworld.com/article/3147892/one-autonomous-car-will-use-4000-gb-of-dataday.html`.

[28] Monitoring amazon msk with amazon cloudwatch - amazon managed streaming for apache kafka. `https://docs.aws.amazon.com/msk/latest/developerguide/monitoring.html`.

[29] Markus Kuderer, Shilpa Gulati, and Wolfram Burgard. Learning driving styles for autonomous vehicles from demonstration. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2641–2646. IEEE, 2015.

[30] Martn Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Man, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Vigas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2015. URL `http://download.tensorflow.org/paper/whitepaper2015.pdf`.

[31] tensorframes. Tensorflow wrapper for dataframes on apache spark. `https://github.com/databricks/tensorframes`, 2016.

[32] Dave Steinkraus, Ian Buck, and PY Simard. Using gpus for machine learning algorithms. In *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, pages 1115–1120. IEEE, 2005.