UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona

SUPPORTING MASSIVE MOBILITY WITH STREAM PROCESSING SOFTWARE

Master in innovation and research in informatics: Computer Networks and Distributed Systems

Advisors:
Prof. Albert Cabellos
Jordi Paillisse

Student:
Luis Eduardo Sosa Salazar

Barcelona, April 2020

Abstract

The goal of this project is to design a solution for massive mobility using LISP protocol and scalable database systems like Apache Kafka. The project consists of three steps: first, understanding the requirements of the massive mobility scenario; second, designing a solution based on a stream processing software that integrates with OOR (open-source LISP implementation). Third, building a prototype with OOR and a stream processing software (or a similar technology) and evaluating its performance.

Our objectives are: Understand the requirements in an environment for massive mobility;Learn and evaluate the architecture of Apache Kafka and similar broker messages to see if these tools could satisfy the requirements; Propose an architecture for massive mobility using protocol LISP and Kafka as mapping system, and finally; Evaluate the performance of Apache Kafka using such architecture.

In chapters 3 and 4 we will provide a summary of LISP protocol, Apache Kafka and other message brokers. On these chapters we describe the components of these tools and how we can use such components to achieve our objective. We will be evaluating the different mechanisms to 1) authenticate users, 2) access control list, 3) protocols to assure the delivery of the message, 4)integrity and 5)communication patterns. Because we are interested only in the last message of the queue, it is very important that the broker message provides a capability to obtain this message.

Regarding the proposed architecture, we will see how we adapted Kafka to store the information managed by the mapping system in LISP. The EID in LISP will be represented by topics in Apache Kafka., It will use the pattern publish-subscribe to spread the notification between all the subscribers. xTRs or Mobile devices will be able to play the role of Consumers and Publisher of the message brokers. Every topic will use only one partition and every subscriber will have its own consumer group to avoid competition to consume the messages.

Finally we evaluate the performance of Apache Kafka. As we will see, Kafka escalates in a Linear way in the following cases: number of packets in the network in relation with the number of topics, number of packets in the network in relation with the number of subscribers, number of opened files by the server in relation with the number of topics time elapsed between the moment when publisher sends a message and subscriber receives it, regarding to the number of topics.

In the conclusion we explain which objectives were achieved and why there are some challenges to be faced by kafka especially in two points: 1) we need only the last location (message) stored in the broker since Kafka does not provide an out of the box mechanism to obtain such messages, and 2) the amount of opened files that have to be managed simultaneously by the server. More study is required to compare the performance of Kafka against other tools.

Acknowledgement

I would like to thank the Polytechnic University of Catalonia for let me be part of its family and of part of this alma matter.

To my family because even in the distance they always motivate me to keep going farther.

To my Tutors Albert Cabellos and Jordi Paillissé for guide me and provide me with all the required knowledge to cover this thesis.

To all my friends , especially to my incredible friend Eva Jannotta for providing amazing support.

Thank you all.

Contents

Abstract 1

Acknowledgement 3

Table of contents 5

List of Figures 6

List of Tables 7

1 Introduction 8

1.1 Objectives of the thesis 9

2 State of the art 10

3 Locator/Identifier Separation Protocol (LISP) 11

3.1 Mapping System 11

3.2 Use cases 13

4 Kafka 14

4.1 Alternatives to Kafka 15

4.2 Push notifications vs Publish subscribe 16

5 Proposal design 19

5.1 Proposed architecture 19

5.2 Consumer groups 20

6 Evaluation of the proposal 22

6.1 Configuration of the server, publisher and subscriber 22

6.2 Experiments 23

6.2.1 Number of packets in the communication vs topics 23

6.2.2 Number of open files vs number of topics using one publisher 27

6.2.3 Number of open files vs number of subscriber 27
6.2.4 Delay between publish and data reception 28

7 Conclusion **31**

8 Future work **32**

Bibliography **33**

Annexes **36**
8.1 Jaas file configuration for communication Zookeeper - Broker and inter-broker communication 36
8.2 Zookeeper jaas file configuration 36
8.3 Access control list configuration 37
8.4 Server configuration file 37

List of Figures

3.1 Example Mapping system. 12

5.1 Proposed architecture. 19
5.2 Consumer groups 21

6.1 Server and clients hardware 23
6.2 Number of packets in network vs topics. 24
6.3 Number of packets in network vs Subscribers. 25
6.4 Number of packets in network,100 topics. 26
6.5 Number of opened files by the server vs number of topics. 27
6.6 Number of opened files by the server vs number of subscribers. 28
6.7 Communication time between publisher and subscriber. 29
6.8 Communication time between publisher and subscriber varying number of subscribers. 30

List of Tables

4.1 Alternatives for kafka 17
4.2 Alternatives for kafka 18

1. Introduction

For the past 10 years, the educational sector and companies have been supporting the development and research of Software Defined Networking (SDN). It had shown itself as a promising alternative for future networks, specifically in fields like security, managing and massive mobility.

In SDN there are two main components: 1) the control plane which provides processes to determine the best routing path (the path that the packet should follow), and 2) the data plane, which manages how to forward those packets between the different hops in the paths. For massive mobility, it is relevant to manage when devices switch the network interface, for example, a mobile that switches from 4G to Wifi, in such event, the device should re-establishes the connection but using a different IP address. Users will notice some delay, such as video freezing or voice interruptions. One of the SDN protocols that support massive mobility is LISP [2].

LISP (Locator/ ID Separation Protocols) is a protocol that lets us decouple the original IP address of the devices from its location in order to be able to identify nodes ignoring the location of such devices. LISP has different components: RLOC or routing locator, ITR or Ingress Tunnel Router, ETR or Egress Tunnel Router, xTR and the mapping system. The Mapping System plays the role of a distributed database, where it stores the mapping between the addresses used to uniquely identify nodes (EID), and the addresses assigned topologically to the inter-domain network interfaces (RLOCs)[1].

For massive mobility, LISP lacks a straightforward mechanism to communicate changes on the endpoints devices. For example, if a mobile switches the network interface, there is no simple way to propagate such notifications to those devices with which communication had been previously established. Location data could be transferred between nodes as simple messages. An alternative to improve such notifications could be achieved using a publish-subscribe broker.

In today's market we can find different message brokers capable of providing security and high availability simultaneously. Between such products we will highlight Apache Kafka. Kafka is a broker message broadly used for the industry as a machine-to-machine

communication mechanism where the publishers (the entity that generates the message) send messages to Apache Kafka nodes which are later consumed by subscribers using communication patterns like publish-subscribe.

The main goal of this thesis is to understand how Apache Kafka can fulfill the requirements of Network State Databases for the context of massive mobility, implemented as a publish/subscribe mechanism.

1.1 Objectives of the thesis

The main purpose of this research project is to :

- Understand the use cases and the requirements in an environment of massive mobility.
- Research and understand the architecture of Apache Kafka.
- Analyze and research similar message brokers.

  These tools should fulfill most of the next requirements:
  Reliability: Guarantee successful delivery of the messages.
  Authentication: It is used to confirm the identity of the user.
  Access Control: Limit the access of read/write on specific topics for a specific user.
  Integrity: The messages are not modified for other entities.
  Last message mechanism: In our use case, we only are interested in the last message of the queue. It is important to know if the message broker has this capability to avoid consuming inaccurate information.

- Determine whether or not Kafka satisfies the requirements from 2 different point of view:
  Qualitative: Analyze if Kafka is able to fulfill the different project requirements.
  Quantitative: Execute experiments to measure the performance of Kafka in a controlled environment.
- Based on the results, determine whether Kafka suits the use cases and if not, give the reasons.

2. State of the art

SDN has been proposed as the technology for solving problems (in a programmatically way) of network management not only for Enterprises but for Cloud providers and Academias. In the educational sector, Openflow[18] is being used as “a way for researchers to run experimental protocols on their networks”,by giving access to the forwarding plane of a network switch or router over the network.

One of the challenges faced by SDN is the creation of protocols for their respective use cases and how such messages will be transported over all the network to notify all the elements. ”Challenges in this area are to find the appropriate control protocol for the specific scenario out of different protocols and protocol versions, and the appropriate forwarding elements that support this protocol”[19]. In addition to those challenges, managing all the information that is running in the network in a centralized way could generate a scalability problem. This could be solved ”by implementing a centralized controller as a distributed system where the contained information has to be maintained consistently”[19].

If we focus on massive mobility, we will need, first, a protocol specifically designed for managing the location of the devices and second, a tool to store or keep track of any change in the network, for example, the destination and source of the packets that need be forwarded in the network. It is required that these tools are able to scalate horizontally and vertically. To solve the first challenge we proposed to use LISP[2], a SDN protocol that decouples location of the identity (more information about LISP will be provided in the next chapters). For the managing and scalable challenge, there are tools like ONOS[20] which is “an experimental distributed SDN control platform motivated by the performance, scalability, and availability requirements of large operator networks” or ONIX[21] “a platform on top of which a network control plane can be implemented as a distributed system”.

3. Locator/Identifier Separation Protocol (LISP)

"The Locator/ID Separation Protocol (LISP) decouples identity from location on current IP addresses by creating two separate namespaces, Endpoint Identifiers to identify hosts and Routing Locators to route packets"[2]. LISP has different components: RLOC or routing locator, ITR or Ingress Tunnel Router, ETR or Egress Tunnel Router, xTR and the mapping system. The Mapping System plays the role of a distributed database, where it stores the mapping between the addresses used to uniquely identify nodes (EID), and the addresses assigned topologically to the inter-domain network interfaces (RLOCs)[1].

3.1 Mapping System

The mapping system is one of the critical elements of the architecture LISP. The main function is to relate the EID with the RLOC, which means translate an identifier to a location. To request the location of an EID the call Map-Request is executed. When the server receives this petition it looks for the Node that has the location of the EID mentioned. After this a Map-Reply instruction is executed, sending back to the ITR the requested locations. It could be different locations because the EID could be associated with more than one RLOCs [17].

In the figure 3.1 we can appreciate a basic example of how the mapping system works. The host identified by **AA:AA:AA:AA:AA:AA** needs to send a message to the host identified by **BB:BB:BB:BB:BB:BB**. When the message is sent, it is captured by one of the two Egress/Ingress tunnel routers (xTR1 or xTR2), then this xTR1 sends a map request to the Mapping System to obtain the RLOC where the host **BB:BB:BB:BB:BB:BB** is located. Finally the mapping system answers with a set of RLOCS where the EID is located **3.0.0.1/32**.

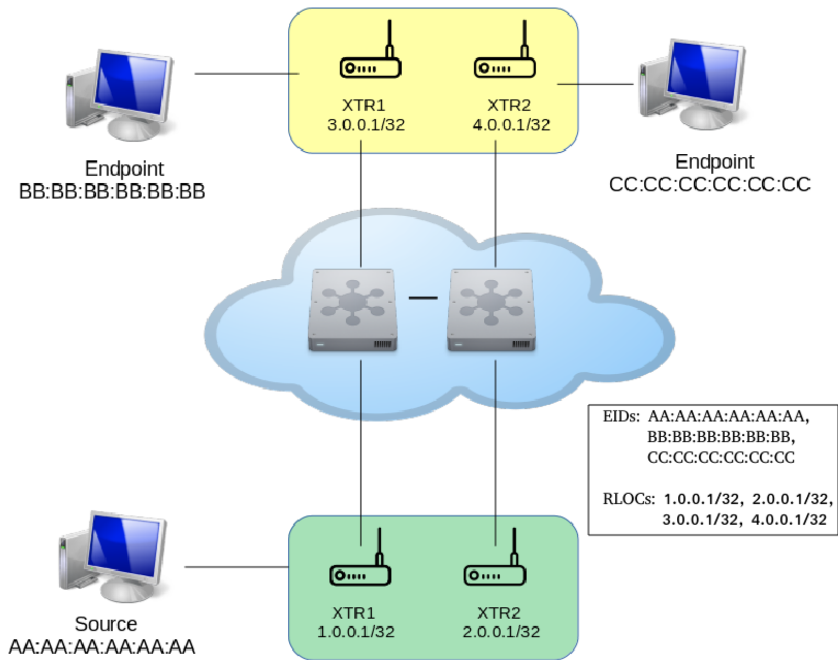

Figure 3.1: Example Mapping system.

We want to use Kafka to store the information regarding the location of the devices. At the same time, this tool will be able to notify all the subscribers if any device has changed its network interface. Sections 4.3 and 5 contain more information with respect to Apache Kafka and how we propose to manage these resources.

3.2 Use cases

Nowadays we could see massive mobility in almost every place we look, when we are driving in the highway and our cellphones changes from one antenna to another to keep us connected to the network, when we leave home and our devices disconnect from our WIFI but almost instantaneously our devices switch to start using 4G to keep us communicated with external services, etc.

In addition to the previous examples, we can find other places where massive mobility could play an important role. In UAV or unmanned aerial vehicles , better known as drones, could help us to improve our live in different ways, “aerial base stations to enhance coverage, capacity,reliability, and energy efficiency of wireless networks, cellular-connected UAVs can enable several applications ranging from real-time video streaming to item delivery” [22].

Smart cars need to be constantly connected to the network in order to offer to users information that could help them taking the best route toward their destination or to inform them if there is any kind of accident in their way to home. For example, through the use of LTE, Car2x “ is a method for data exchange between vehicles and infrastructure units to increase traffic safety and efficiency” [23].

Virtual machines migration is a common mechanism used in datacenter to move VMs between different location, enterprises could decide to move VMs between datacenter for example if they want to reduce cost by lowering the energy consumption or if they want to distribute work load. [24]

4. Kafka

Apache Kafka is a distributed streaming platform which has three main capabilities. First, it uses the communication pattern publish-subscribe to stream records as it is used in similar message queues. Second, it processes the messages as soon as they arrive. Third, a fault-tolerant store mechanism is used to save the streams of records. The common use of Apache Kafka is in applications for real-time streaming data pipelines or applications that transform or react to the streaming of data. For inter-communication, TCP protocol is utilized for servers, zookeepers and clients. [3]

Messages are published to a category better known as topic. Consumers subscribe to these topics, a topic could have zero or more subscribers. Every topic has one or more partitions. A partition is an immutable, ordered sequence of messages. Every message has a sequential identifier called offset. Partitions are distributed between Kafka servers, every partition has one leader server which handles all read and write requests. Followers just replicate the data of the master.[3]

Apache Kafka has three components to add security to the server. SSL/TLS communication, which encrypts all transported data between publisher, subscriber and server. For authentication, it has SSL and SASL. SSL authentication means using a certificate to authenticate clients, this certification is signed by a certificated authority. SASL or simple authorization service layer is used as alternatives to authenticate users. These are SASL alternatives supported by Kafka: SASL PLAINTEXT, SCRAM, SASL GSSAPI and SASL OAUTHBEARER. Plaintext is the classic user and password mechanism for authentication, the user and password are stored in a file on the Kafka brokers. The SASL Scram is the mechanism that we will be using in all our experiments, with the idea to combine a user and password together with a challenge called salt, where the password is hashed and stored in the zookeeper. With this mechanism you could add or modify user passwords without rebooting brokers. SALS GSSAPI is the well known mechanism called Kerberos, and the final mechanism is the SASL OAUTHBEARER. As its name says, it lets Apache Kafka use oauth2 tokens. Access control list will be covered in another section.

4.1 Alternatives to Kafka

Currently, in the market, it is possible to find a great number of tools for managing queues of messages or message brokers. In this section we will describe the tools that were considered capable of fulfilling the requirements of the mapping system in the Kafka protocol.

RabbitMQ is a queue manager or message broker that implements different message protocols like Advanced Message Queuing Protocol (AMQP[10]) or Streaming Text Oriented Messaging Protocol (STOMP). In the implementation of the protocol AMQP 0-9-1 it uses TCP for reliable delivery and for secure connection it supports TLS. In RabbitMQ there are 2 main mechanisms for authentication, a regular user password pair and an Internet X.509 [8]. Additionally, there are plugins that expand the capabilities of authentication and authorization, for example, Lightweight directory access protocol (LDAP) plugin. The exchange type fanout is used in RabbitMQ when the messages should be sent to all the subscribers. There is not a direct way to obtain the last message in the queue. The messages can be moved to a disk to persist them.[9]

ZeroMQ "gives you sockets that carry atomic messages across various transports like in-process, inter-process, TCP, and multicast. You can connect sockets N-to-N with patterns like fan-out, pub-sub, task distribution, and request-reply"[11]. It implements different transport protocols like multicast, TCP and web sockets. There is a plain user/-password mechanism for authentication, this could be used in safe internal networks, additionally the application could be adapted to use CurveZMQ which is an authentication and encryption protocol for ZeroMQ [12]. There is an option that could be activated (ZMQ_CONFLATE) to keep only the last message in the queue. For access control there is not a simple mechanism, a firewall should be used to filter connections.

Nat.io "is an open source, lightweight, high-performance cloud native infrastructure messaging system. It implements a highly scalable and elegant publish-subscribe (pub-/sub) distribution model. The performant nature of NATS makes it an ideal base for building modern, reliable, scalable cloud native distributed systems"[13]. In Nats.io, TCP is used in connection to keep reliability of delivery. When a message is ready to communicate with the clients, the server "fires and forgets", meaning when a message is ready to be sent, the server forgets about it and trusts in TCP that the message will be delivered correctly. There are several mechanisms in nats to authenticate. Token Authentication, username/password credentials, certificates, etc.Nats supports authorization using permissions, for every permission related to a subject the administrator of the server can

define to which subject the user can publish to or subscribe to. Publish-subscribe pattern could be used in Nats.

EMQX "Broker is a massively scalable, highly extensible distributed MQTT message broker written in Erlang/OTP. It is open source and is highly extensible. You can write your own plugins to support proprietary protocols at TCP/UDP layer, store data into a database or integrate with an external system" [15]. It implements MQTT protocol which uses TCP for communication.Other protocols could be added if they are implemented as plugins. For authentication we can use:TLS certification, simple user and password, http authentication invoking customized HTTP API, Lightweight Directory Access Protocol (LDAP) and different databases-based authentication like PostgreSQL and MYSQL. It has an Access control list mechanism that allows us to limit which topics the publisher and subscriber can send or read messages. EMQX uses the pattern publish-subscribe for communication. As we mentioned before in our requirements, it is needed to keep the last message for every topic in that way every new subscriber could read such message. EMQX implements this through a mechanism called Retain Message, it is a feature of MQTT 5.0.

4.2 Push notifications vs Publish subscribe

Broker messages let us to send messages to subscribers using different communication patterns, in this case we will highlight two of them, the pattern push notification and the pattern publish-subscribe. Push notification is broadly used in mobile technology, specifically when it is needed to notify the final user about an action that should be executed. The final user could be a person or an application. In this pattern the client just waits for messages coming from the server. The second pattern is called publish subscribe, and this is the pattern used in Kafka. One of the main reasons why Kafka uses this pattern is because a "push-based system has difficulty dealing with diverse consumers as the broker controls the rate at which data is transferred"[5]. That means that the subscriber could be overwhelmed by the server if they are not able to keep the pace of consuming messages. With publish-subscribe the consumer defines the rate of messages to be consumed, and it is possible to have different subscribers with different capabilities.

Tool Requirement	**Kafka**	**RabbitMQ**	**ZeroMQ**
Reliability	TCP, ACK property for consumer	TCP.	TCP, If need more mechanism, they should be implemented by user
Authentication	SASL(Simple Authentication and Security Layer), plain user/ password, kerberos, oauth2	Lightweight Directory Access Protocol (LDAP), Simple authentication security layer . X.509 rfc	Plain user/password, CURVE
Access Control	It uses ACL (Access control list), permission on read, write on specific topic.	Lightweight Directory Access Protocol (LDAP), Connections, exchanges, queues, bindings, user permissions, policies belong to virtual hosts	No simple solution ,Firewall, encrypt connection where you are the only one with access.
Integrity	SSL/TOKEN	SSL/TLS	SSL
Communication Pattern	Publish-Subscriber	Publish-subscriber (fanout)	Publish-subscriber
Last message mechanism	Track position of the message using offset	Can store the message	It is called ZMQ_CONFLATE The idea is to maintain only the last message in the queue.

Table 4.1: Alternatives for kafka

Tool Requirement	Nats	MQTT Protocol EMQX
Reliability	Rely on TCP, no persistence of messages, fire and forget	Implement MQTT protocol (TCP)
Authentication	User/ (encrypt)password, Create an account that could have several users. Token mechanism	It is possible to create users and passwords.
Access Control	List of subjects to write or read. Or where to subscribe, Subjects denied.	MQTT uses ACL. It is possible to control who is pushing or subscribing to a specific topic. Additionally in MQTT is possible to use wildcard to subscribe to a group of topics
Integrity	TLS	TLS
Communication Pattern	Publish-subscribe	Publish-subscribe
Last message mechanism		MQTT keeps the last message of a topic by setting the retained message flag to 1. Only one message is retained per topic

Table 4.2: Alternatives for kafka

5. Proposal design

This chapter will describe the design of the architecture LISP integrating Kafka as a mapping system.

5.1 Proposed architecture

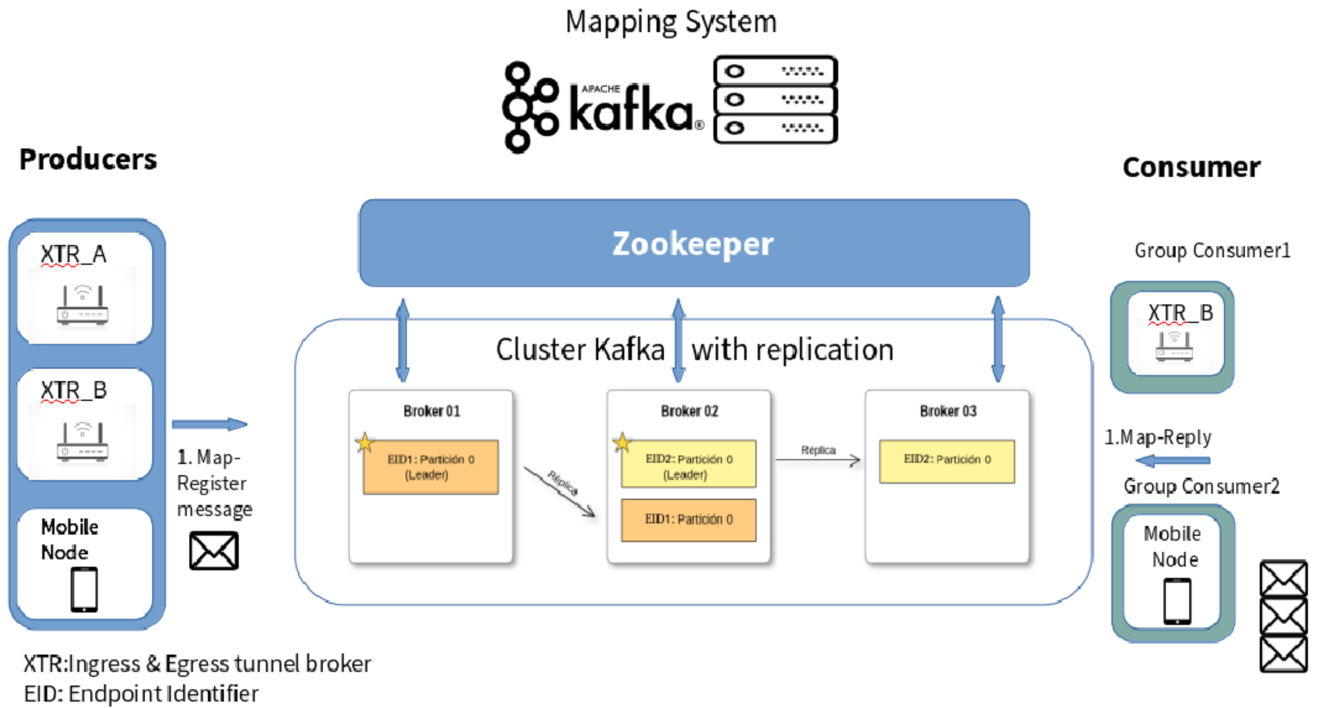

Figure 5.1: Proposed architecture.

Producers: Entities that create or produce the messages to be consumed. The entities are xTRs or mobile nodes. These entities execute the instruction map-register of the protocol LISP, this instruction is used to registry an EID-to-RLOC mapping [1].

Consumers: Entities that process or consume the messages sent by the producers. xTRs and mobile nodes are consumers. They execute the instruction map-request of the protocol LISP. Map-request means a signaling message that is sent to obtain the resolve EID-to-RLOC mapping [1].

Zookeeper: Zookeeper is a coordinator of services (high performance) for distributed applications. The main function is to expose the common services like configuration, naming, synchronization, list of services, etc. Zookeeper provides as well a mechanism that implements consensus, leader election, and presence protocols in a simple interface that you don't need to implement from zero.[4]

Kafka broker: Better known as a kafka-server, it receives messages coming from producers and stores them into disk using an id for every registry, this id is called offset. It allows the consumer the capability of downloading the messages by offset, partition and topic. Every kafka-server has a configuration file where it is possible to define the properties that will determine the behavior of the server[3].

Consumer group: This concept is used in Kafka in order to implement two different patterns. If we have multiple consumers inside of the same consumer group, we will be implementing the pattern "competing consumers" which means that all the messages are distributed between all the members of the group. In contrast, if every consumer has a different consumer group associated with the same topic, all the messages that have been sent to the topic will be received equally for all the members of the different groups; this pattern is known as "publish/subscribe".

5.2 Consumer groups

The way that messages are consumed by the subscribers in a kafka environment will vary depending on how they are configured. They are two main configurations. The first configuration occurs when members of the same consumer group will be competing between them to process the messages. This means that for all the members belonging to a consumer group that have subscribed to a partition topic, the message will be delivered to only one of them and it won't spread to the other members of the consumer group.

The second configuration, which is the configuration that we are proposing to be used in the architecture, consists of separating every subscriber in a different consumer group, through this we can ensure that consumers behave as the pattern publish-subscribe.

Putting all the subscribers in different consumer groups means all the messages received on a topic partition will be sent to all the subscribers. As we can see in the figure 5.2, consumer groups will be composed by one xTR or Mobile Node.

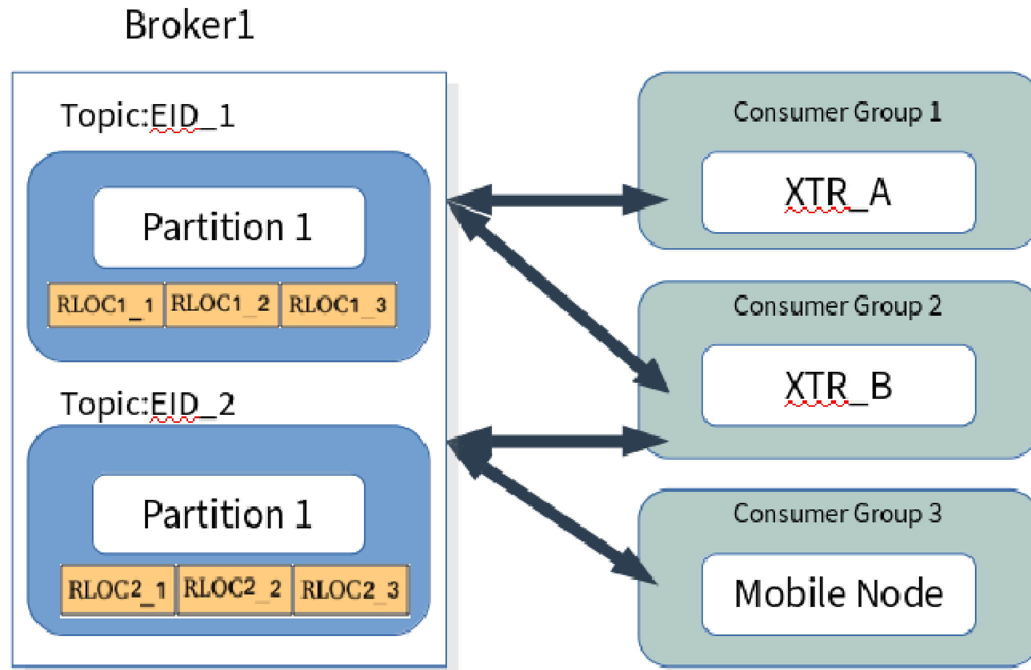

Figure 5.2: Consumer groups

Every Topic represents an EID, and in our case, we only need one partition per topic, This will make it easier to manage the order of incoming messages and to easily spread those messages between all the subscribers.

6. Evaluation of the proposal

6.1 Configuration of the server, publisher and subscriber

Kafka 2.12-2.3 was the version installed for the experiments, and it can be downloaded from the Apache Software Foundation web page [5]. Kafka was installed in a virtual machine running Ubuntu 18.04 with 5 physical processors 2.1GHz, 2 cores and 4 threads . 12GB of RAM.

Security for the communication between Zookeeper and Kafka Broker: Scram or Salted Challenge Response Authentication Mechanism is a mechanism for secure authentication that provides some advantages over the traditional Digest mechanism because the Zookeeper stores all the data related to the password in an irreversible format, which means the password is hashed[6].

Communication between the Zookeeper and the Broker will be using the traditional Digest Login Mechanism. The communication between Kafka-servers and Kafka broker- -clients will be using the Scram Module.Zookeeper needs a jaas configuration file where the mechanism used for communication (between the broker and the Zookeeper) is described. See annexes for more details about the specific code that should be added to the jaas files.

Kafka provides a simple command line to manage how and who can access the different resources. We have defined two different users in the previous configuration (jass files), now we have to create those users in the Zookeeper. See annexes and sub-section control list, for more details. The file server.properties located in the kafka broker should be modified if we want to use scram over plain text communication, and we have to set some properties. See annexes, section Server configuration file for more details.

The publisher was installed in a virtual machine different to the server and subscriber. It had 2 physical processors, 2 cores and 2 threads and 4 GB of ram. The operative system used was Ubuntu 18.04.4 LTS. The subscriber was installed in a virtual machine different to the server and publisher. It had 2 physical processors, 2 cores and 2 threads

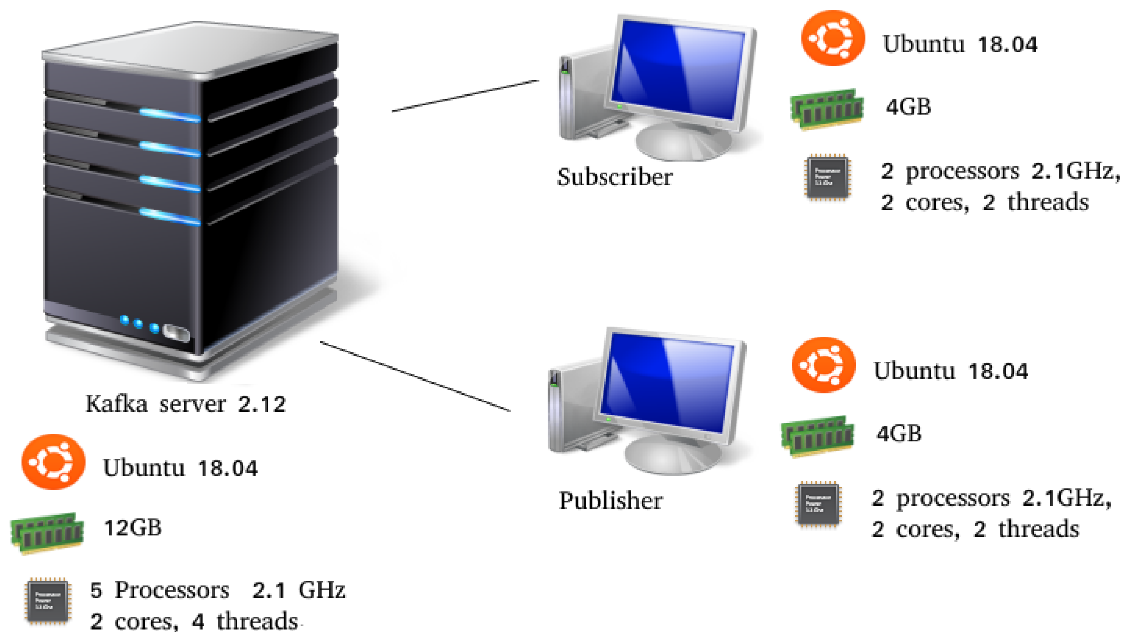

Figure 6.1: Server and clients hardware

and 4 GB of ram. The operative system used was Ubuntu 18.04.4 LTS.

6.2 Experiments

Experiments were executed using the previously described configuration for the kafka server, publisher and subscriber (see figure 6.1 for reference). Packets sent by the publisher, number of opened files by the server and the time elapsed between the moment that a publisher sends a message and it is received by the subscriber are some of the experiments that we are going to review in this section.

6.2.1 Number of packets in the communication vs topics

The main objective of the first experiment was to measure the number of sent and received packets by the publisher and subscriber respectively. The first experiment was made without subscribers so, we are measuring the number of received and sent packets needed to transmit 1 message to the server for all the topics. In this case we wanted to see if there is a relationship between the number of packets in the network and the number of topics. We varied the number of topics from 100 to 300, increasing them by 50 per experiment,

and we repeated each experiment 15 times. The tool used for capturing the number of packets was **tshark**[14]. The next filters were applied. The variable **SOURCE** and **DESTINATION** contains the ip address of the publisher and the server respectively. The variable **INTERFACE** as its name mentiones, it contains the name of the network interface used to capture the messages. All the captured packets belong to the layer 3 (L3).

```
tshark  -i $INTERFACE -Y " (ip.src==$SOURCE and ip.dst==$DESTINATION)"

tshark  -i $INTERFACE -Y " (ip.src==$DESTINATION and ip.dst==$SOURCE)"
```

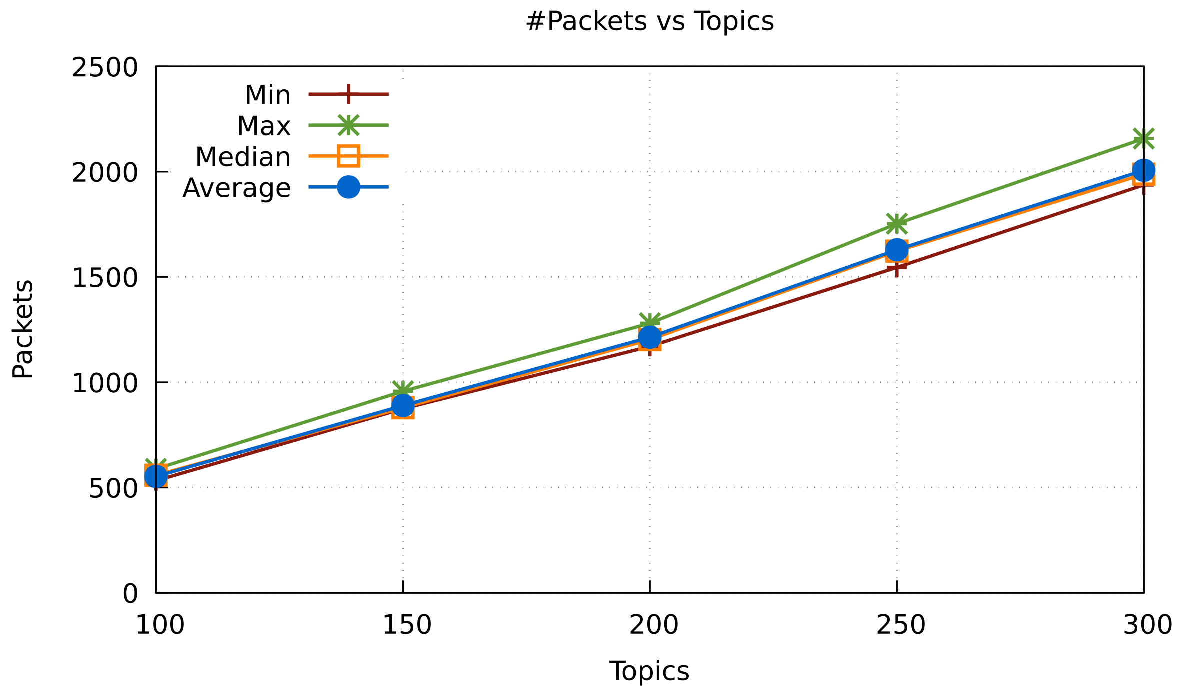

Figure 6.2: Number of packets in network vs topics.

In Figure 6.2 we can see a linear relation between the number of topics and the number of packets in the network used in the communication between the server with the publisher. As we increase the number of topics, there is an increase in the number of captured packets. Using this graphs we can estimate the impact of packets on the network when the number of topics is increased.

In the next experiment we wanted to see the impact on the network of increasing the number of subscribers. In this case we increased the number of subscribers from 1 to 20. All the subscribers were doing pools until they received the first message, this is when

the experiment ends.

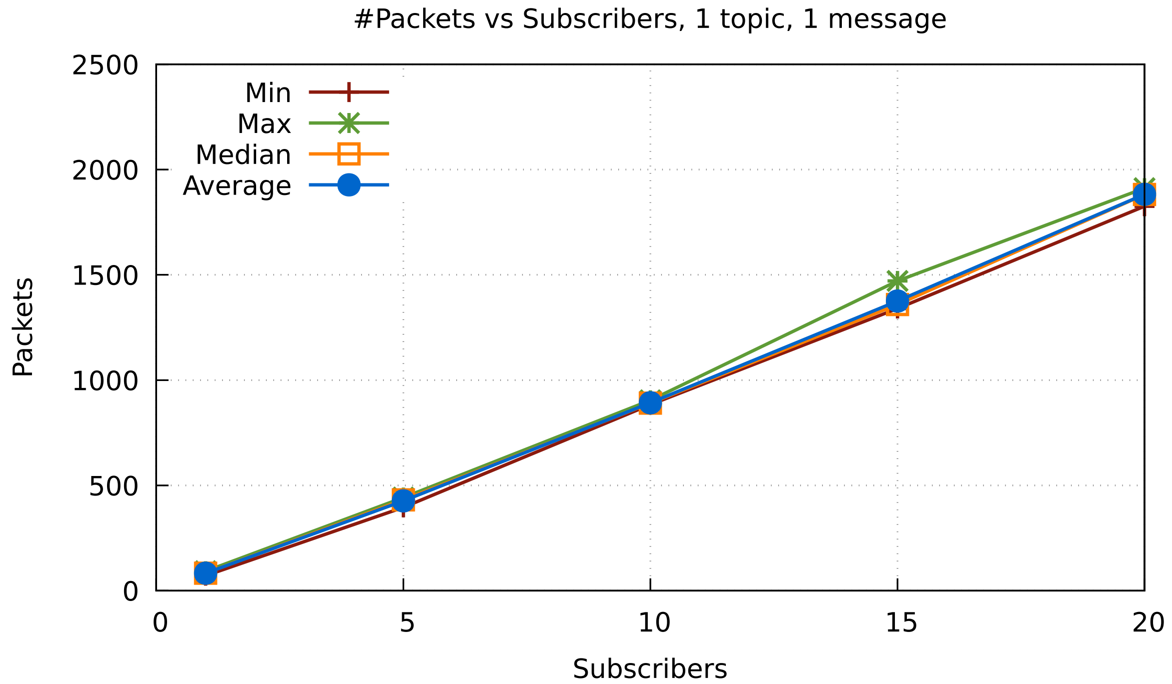

Figure 6.3: Number of packets in network vs Subscribers.

As we can observe in Figure 6.3, there is a linear relationship between the number of transmitted packets by the Kafka server and the number of subscribers. As we increased the number of subscribers there was an increase in the number of packets captured. From this graph we can give an estimation of how the network will be stressed if we increase the number of consumers. In this experiment as mentioned before, all the consumers were in different consumer groups and they were only subscribed to one topic. Every experiment was repeated 15 times.

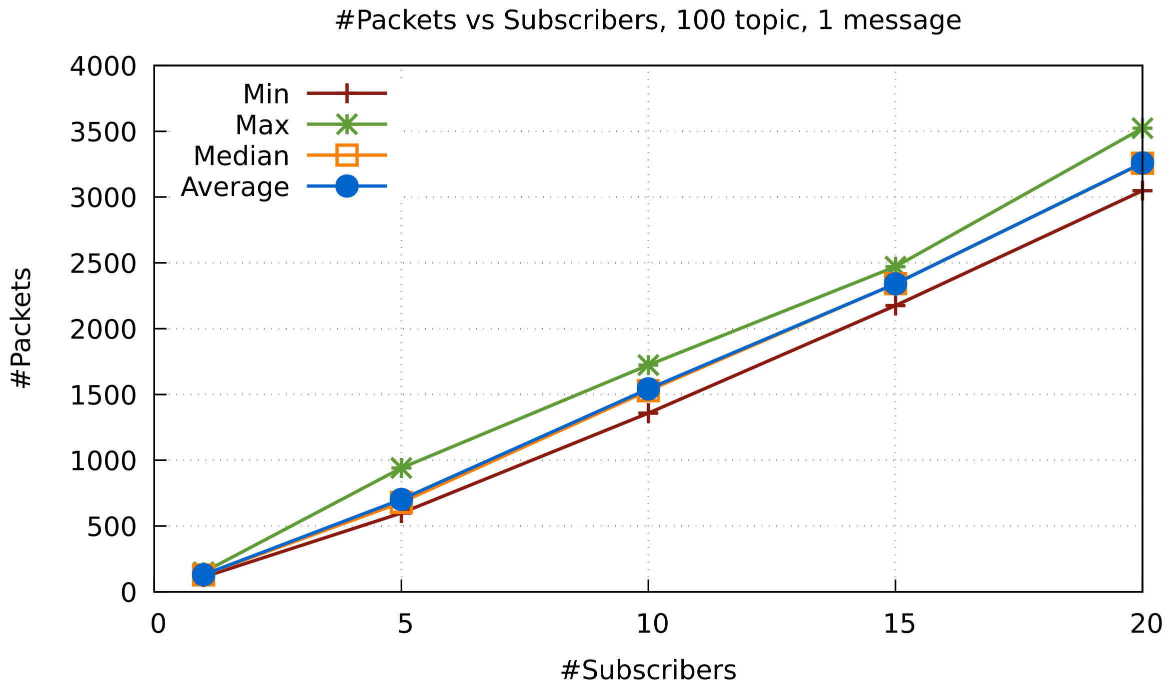

Figure 6.4: Number of packets in network,100 topics.

Taking in considerations the results of the previous experiment, we wanted to know if the number of topics affect the number of packets in the network, In this case, instead of having the consumers subscribe to 1 topic, we subscribe the consumers to 100 topics. All the subscribers consume the first message from all 100 different topics, the results are shown in the graph 6.4. There is still a linear relationship between the number of subscribers and the topics. As we expected, in this experiment we captured more packets than in the previous experiment where the subscribers were only consuming from 1 topic. The maximum number of packets was 1912 in that case. In this new experiment the maximum number of packets was 3524, an increase of 84% on the number of messages.

6.2.2 Number of open files vs number of topics using one publisher

Here we were looking for a relationship between the number of opened files by the kafka server and the number of topics. As we can see in the next graph there is a linear relationship between these two variables. It is important to know the number of opened files by the kafka server because as mentioned previously, the topics in the proposed architecture will represent the ID of the different devices. Once we initiate the process for sending the message from the publisher to the server, the script located in the server starts collecting the statistics of opened files.

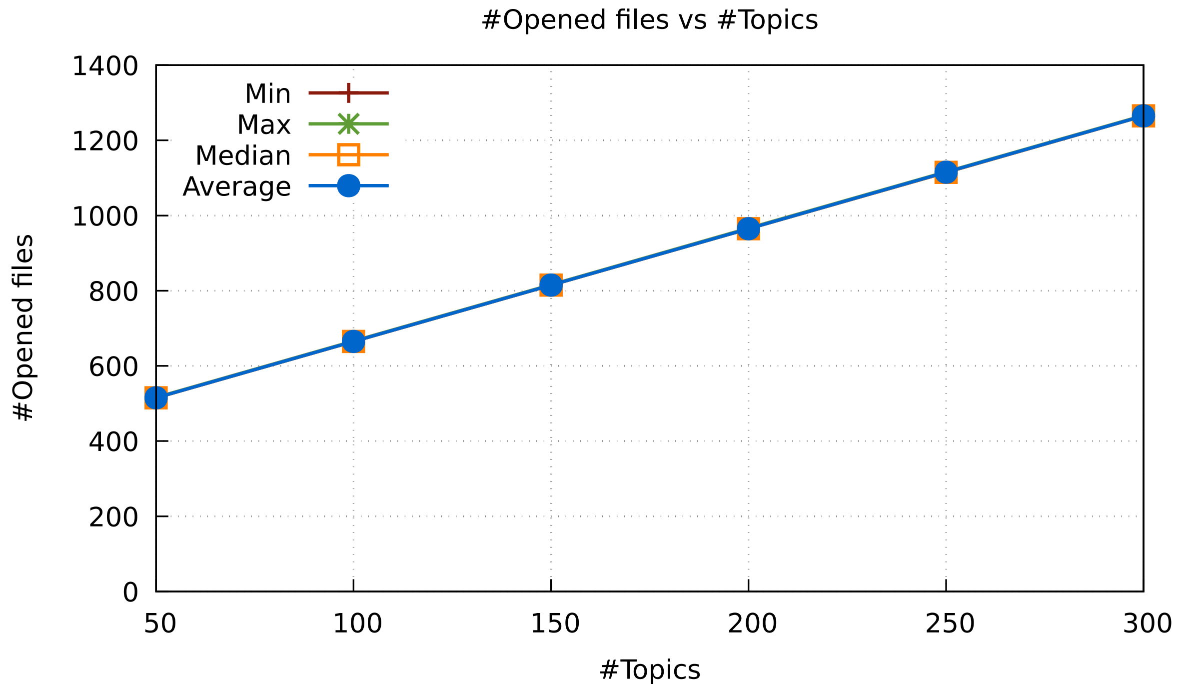

Figure 6.5: Number of opened files by the server vs number of topics.

As we can see in the graph 6.5, there is a linear relationship between the number of opened files by the kafka server and the number of topics. This means that if we add more EIDs to the platform we could estimate the number of opened files that the kafka server will have to manage.

6.2.3 Number of open files vs number of subscriber

In the next experiments we wanted to evaluate how the number of subscribers affect the number of opened files by the kafka server. For this we subscribed from 1 to 20 subscribers

to the kafka server. They consumed messages from 100 topics, a static number that will not vary during the experiment.

It is possible to see in the figure 6.6 that increasing the number of subscribers does not affect the number of opened files by the kafka server. We could increase the subscribers without having a noticeable impact on the performance of the server regarding the number of opened files.

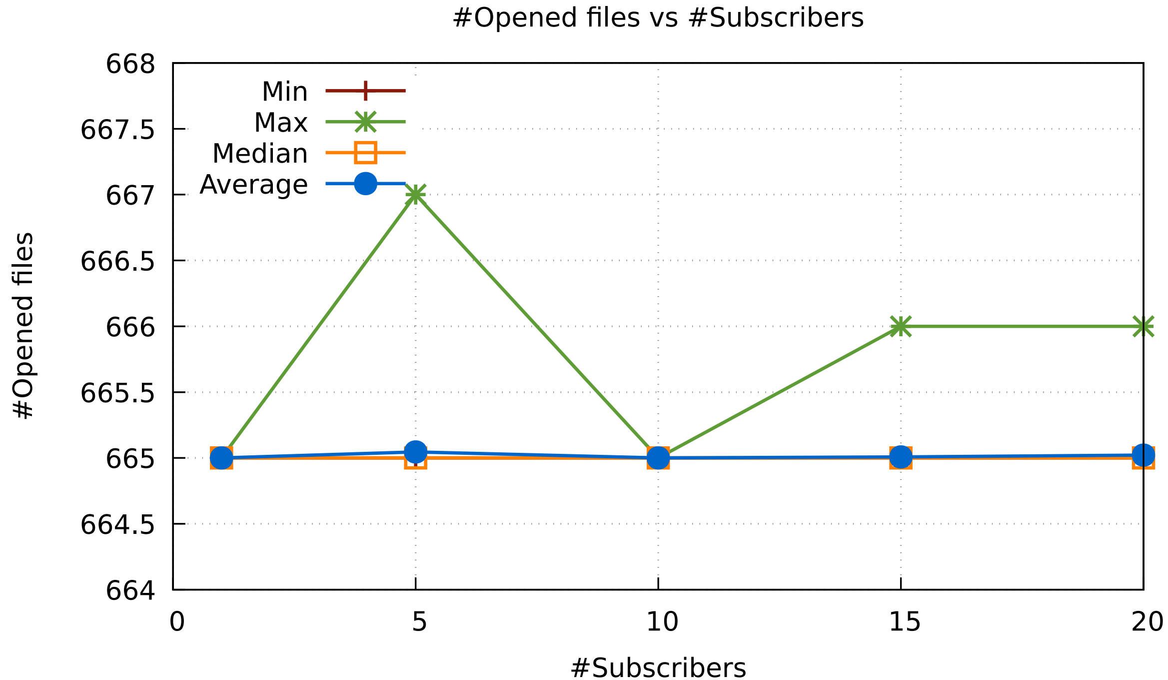

Figure 6.6: Number of opened files by the server vs number of subscribers.

6.2.4 Delay between publish and data reception

In these next experiments we measured the time elapsed between the moment when the publisher sends a message to the kafka server until it is processed by the subscriber. Server, publisher and subscriber, as mentioned before, are running in different virtual machines. We increased the number of topics by 50 in every experiment, starting with 100 topics and finishing with 300 topics. Only one subscriber and one publisher were used. The process of sending the messages could be summarized as: Publisher open connection with the kafka sever, it writes 1 message for every topic, close connection. At the same time there is an instance of the Subscriber waiting for messages.

The figure 6.7 shows that there is a linear relationship between the number of topics from which the subscriber is consuming messages and the time it takes for the consumer

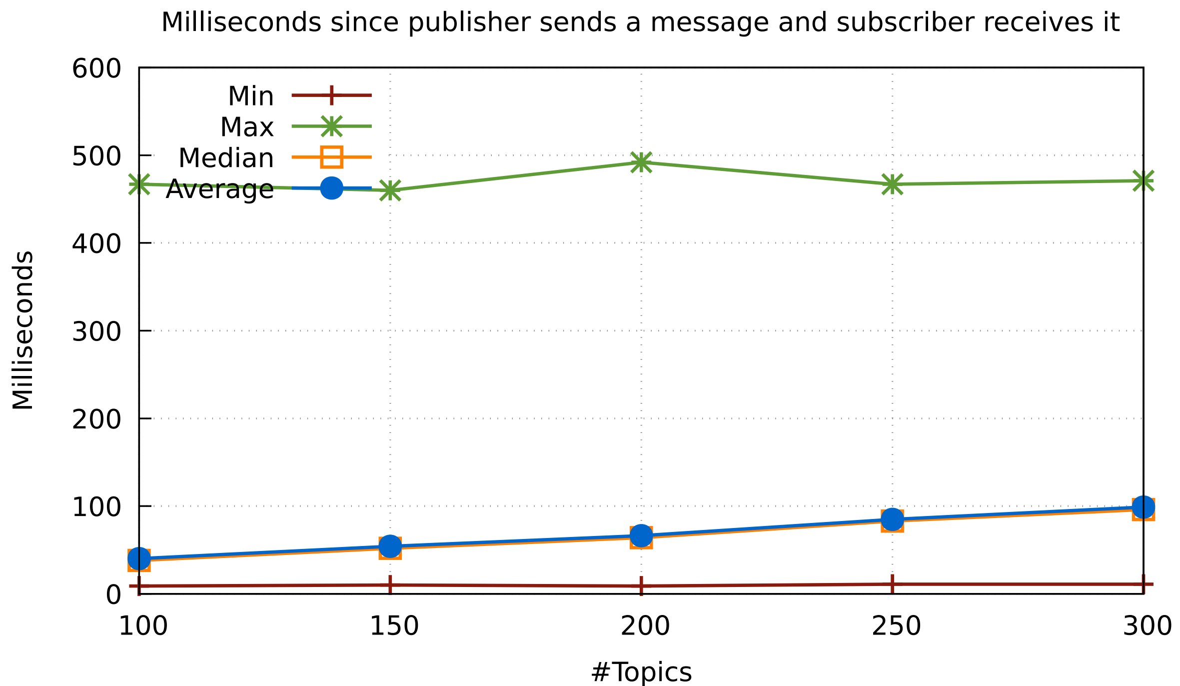

Figure 6.7: Communication time between publisher and subscriber.

to receive the first message. As we increased the number of topics there was an increment in the time elapsed between the instant when the publisher sends the message and the moment that it is received by the subscriber.

We executed a similar experiment but in this case the number of topics was constant, at 1,but the number of subscribers varied from 100 to 700, increasing in order of 100 in every iteration. Every subscriber received the same message. For this experiments we used python and a library called confluent_kafka [25]. The reason to do this is because every consumer was running as a different process which implies create a java virtual machine for every consumer. Every virtual machine was consuming to much memory for a simple consumer for that reason we decide to use a lighter client.

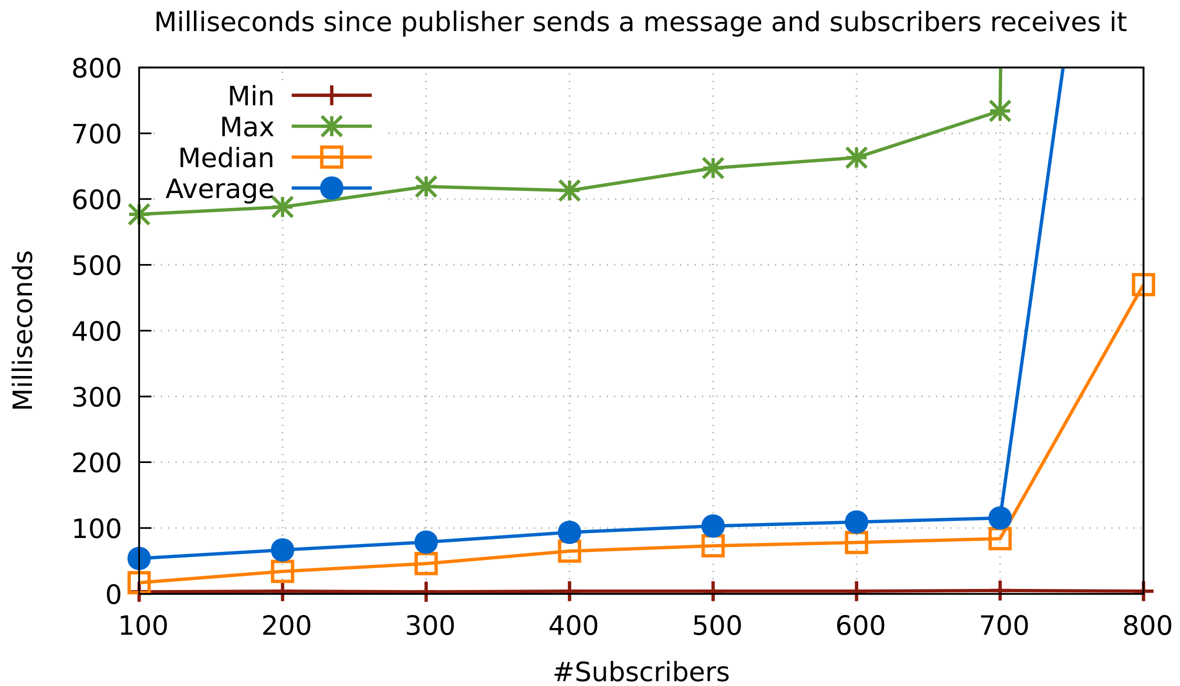

Figure 6.8: Communication time between publisher and subscriber varying number of subscribers.

When we increased the number of subscribers, there was an increase in the milliseconds that it took for the subscriber to process the message. In this case the image represents the worst case scenario where all the subscribers are running in the same machine. Of course, the amount of time that it could take for every subscriber would be smaller if they were running on separate devices. Similar to the previous result, there was a linear relationship between these 2 variables. The peak at the end of the graph was caused when the machine where subscribers were running started to run out of memory and use the swap memory.

7. Conclusion

As we have observed Apache Kafka fulfills most of the requirements that we have established at the beginning of this thesis but it still lacks some direct mechanisms to obtain the last message of the queue. Despite this, the mapping system could still take advantage of the other capabilities of Kafka.

Kafka uses TCP/IP as protocol to control that the message or packets have been transmitted and received correctly by all the devices. Regarding authentication in Kafka, SCRAM-256 and SCRAM-512 (SASLs) provide a mechanism for avoiding fake authentication against the server. The Hashed password is stored in the Zookeeper after applying a defined number of iterations. Because of this, if someone is able to break through the security of the database and access the data stored on it, it will be difficult to recover the passwords. Scram is secure against replay attack but is not a perfect solution. For Integrity, if there is a man in the middle attack, the attacker could have sufficient information to mount an offline dictionary or brute-force attack and for that reason it is recommended to use it together with TLS[6]

Access control list is totally covered in Kafka, we can limit the access of read/write on a specific topic for a specific user. As we mentioned previously, we are interested only in the last location messages sent for every device, and Kafka does not provide a direct mechanism to obtain that message. If we want to request the last message stored in a queue, it is necessary to execute 2 polls.

In all the experiments related to the transmission time, we obtained a delay lower than a 1 second. In the case where we were testing with 700 subscribers the average time obtained was lower than 200 milliseconds. In all the experiments we observed that kafka was able to scalated in a linear way. These results let us conclude that kafka is a trustworthy tool for scenarios where scalability is desirable.

Regarding the opened files by the server, we observed that there is a linear relationship between the number of topics and the number of files created and opened by the server to control and store data related to the messages. So if we wanted to manage a millions of devices, there will be at least 4 millions of opened files that should be handled by Kafka.

8. Future work

In the market we can find promising tools that could help us to improve the way that xTRs could be notified if any device has change of location. In this thesis we have given a brief review of some of those tools and how the requirements are fulfilled by them. Between all the available tools there is one that I would like to highlight: EMQX. It is open source and it uses TCP for communication, EMQX has a basic user and password mechanism for authentication but at the same time provides the capability to implement our own authentication mechanism. Access Control List could be used to control who is pushing or subscribing to messages. It uses a publish-subscribe pattern and has a last message mechanism to keep the last message received by the broker, this is the main reason that makes EMQX a promising tool. The baseline obtained in the experiments could be used as an initial benchmark to evaluate other message brokers.

Bibliography

[1] Albert Cabellos-Aparicio and Damien Saucez. *An Architectural Introduction to the Locator/ID Separation Protocol (LISP), draft-ietf-lisp-introduction-13.txt.* April 02, 2015. https://datatracker.ietf.org/doc/draft-ietf-lisp-introduction/

[2] Alberto Rodriguez-Natal, Marc Portoles-Comeras,Vina Ermagan, Darrel Lewis, Dino Farinacci, FabioMaino, and Albert Cabellos-Aparicio *LISP: a southbound SDN protocol?.* 2015. IEEE CommunicationsMagazine53, 7 (2015), 201–207

[3] The Apache Software Foundation. *Apache Kafka Documentation.* Last access February 15, 2020. Retrieved from https://kafka.apache.org/documentation/#configuration

[4] The Apache Software Foundation. *ZooKeeper: Because Coordinating Distributed Systems is a Zoo.* Last access February 15, 2020. Retrieved from https://zookeeper.apache.org/doc/r3.5.7/index.html

[5] The Apache Software Foundation. *kafka2.12-2.3.0.* Last access February 18, 2020. Retrieved from https://www.apache.org/dyn/closer.cgi?path=/kafka/2.3.0/kafka_2.12-2.3.0.tgz

[6] Nicolas Williams, *Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms.* Last access February 18, 2020. Retrieved from https://tools.ietf.org/html/rfc5802

[7] The Apache Software Foundation. *Kafka Authorization Command Line Interface.* Last access February 18, 2020. Retrieved from https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Authorization+Command+Line+Interface

[8] Matt Cooper,Yuriy Dzambasow, Peter Hesse,Susan Joseph,Richard Nicholas. *Internet X.509 Public Key Infrastructure: Certification Path Building.* Last access February 29, 2020. Retrieved from https://tools.ietf.org/html/rfc4158

[9] RabbitMQ. *Documentation*. Last access February 29, 2020. Retrieved from https://www.rabbitmq.com/documentation.html

[10] Sanjay Aiyagari, Cisco Systems, Matthew Arrott ,Mark Atwell. *AMQP Advanced Message Queuing Protocol.*. Last access February 29, 2020. Retrieved from https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf

[11] Zeromq *ZeroMQ An open-source universal messaging library*. Last access February 29, 2020. Retrieved from https://zeromq.org/

[12] iMatix Corporation. *CurveZMQ - Security for ZeroMQ*. Last access February 29, 2020. Retrieved from http://curvezmq.org/

[13] Nats.io. *Documentation*. Last access February 29, 2020. Retrieved from https://docs.nats.io/

[14] Wireshark. *Documentation*. Last access March 8, 2020. Retrieved from https://www.wireshark.org/

[15] Emqx. *Documentation*. Last access March 8, 2020. Retrieved from https://docs.emqx.io/broker/latest/en/

[16] Loránd Jakab , Albert Cabellos, Florin Coras, Damien Saucez, Olivier Bonaventure. *Evaluating the Performance of LISP Mapping Systems*. 2012

[17] Loránd Jakab , Albert Cabellos, Florin Coras, Damien Saucez, Olivier Bonaventure. *Evaluating the Performance of LISP Mapping Systems*. 2012

[18] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. *OpenFlow: enabling innovation in campus networks*. 2008. ACM SIGCOMM Computer Communication Review, 2008, vol. 38, no 2, p. 69-74.10.1145/1355734.1355746

[19] Michael Jarschel, Thomas Zinner, Tobias Hoßfeld,Phuoc Tran-Gia, and Wolfgang Kellerer. *Inter-faces, attributes, and use cases: A compass for SDN.Communications*. Magazine, IEEE52, 6 (2014), 210–217

[20] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. *towards an open, distributed SDN OS*. En Proceedings of the third workshop on Hot topics in software defined networking. 2014. p. 1-6.

[21] Teemu Koponen, Martin Casado, Natasha Gude,Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ra-manathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama. *Onix: A Distributed Control Platform for Large-scale Production Networks.* 2010

[22] M. Mozaffari, W. Saad, M. Bennis, Y. Nam and M. Debbah. *A Tutorial on UAVs for Wireless Networks: Applications, Challenges, and Open Problems.* IEEE Communications Surveys & Tutorials, vol. 21, no. 3, pp. 2334-2360, thirdquarter 2019.

[23] M. Mazzola, G. Schaaf, F. Niewels and T. Kurner, *Exploration of Centralized Car2X-Systems over LTE.* 2015 IEEE 81st Vehicular Technology Conference (VTC Spring), Glasgow, 2015, pp. 1-5.

[24] D. Amendola, N. Cordeschi, and E. Baccarelli, *Bandwidth management VMs live migration in wireless fog computing for 5G networks.* in Proc. 5th IEEE Int. Conf. Cloud Netw. (Cloudnet), Pisa, Italy, 2016, pp. 21–26

[25] Magnus Edenhill, Ryan P *Confluent's Python Client for Apache Kafka* Last access February 10, 2020. Retrieved from https://github.com/confluentinc/confluent-kafka-python

Annexes

8.1 Jaas file configuration for communication Zookeeper - Broker and inter-broker communication

The next code should be added in the kafka server jaas config file.

```
Client {
        org.apache.zookeeper.server.auth.DigestLoginModule required
        username=``kafka"
        password=``kafka-secret";
};

KafkaServer{
    org.apache.kafka.common.security.scram.ScramLoginModule required
    username=``kafkabroker"
    password=``kafkabroker-secret"
    user_kafkabroker=``kafkabroker-secret"
    user_client=``client-secret";
};
```

Here we have the admin user **kafkabroker** and its password **kafkabroker-secret**, additionally we are adding one user **client** for managing the server with password **client-server**.

8.2 Zookeeper jaas file configuration

Here we are defining the admin user of the kafka zookeeper.

```
Server {
        org.apache.zookeeper.server.auth.DigestLoginModule required
        user_super=``admin-secret"
        user_kafka=``kafka-secret";
};
```

8.3 Access control list configuration

Use the next commands to create the user kafkabroker and client

```
kafka-configs.sh --zookeeper localhost:2181 --alter
--add-config 'SCRAM-SHA-256=[password=kafkabroker-secret]'
 --entity-type users  --entity-name kafkabroker

kafka-configs.sh --zookeeper localhost:2181 --alter
--add-config 'SCRAM-SHA-256=[password=client-secret]'
--entity-type users --entity-name client
```

Additionally we have to grant access to the topics and the consumer groups[7].

```
kafka-acls.sh
--authorizer-properties zookeeper.connect=localhost:2181
--add  --allow-principal User:client  --operation All
--topic=* --group=*
```

8.4 Server configuration file

```
#Setting kafkabroker user as the admin user of the broker
super.users=User:kafkabroker

#The data will be traveling in plain text.
security.protocol=SASL\_PLAINTEXT

#Exposing the port where we are going to be waiting for messages
#Replace 127.0.0.1 with the IP of the kafka server
listeners=SASL\_PLAINTEXT://127.0.0.1:9092
advertised.listeners=SASL\_PLAINTEXT://127.0.0.1:9092

#Comunication inter broker will be done using plain text
#For authentication it will be required Scram-256.
#Replace 127.0.0.1 with the IP of the Kafka server
security.inter.broker.protocol=SASL\_PLAINTEXT
```

```
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-256
sasl.enabled.mechanisms=SCRAM-SHA-256
advertised.host.name=127.0.0.1
```