Performance Evaluation of DIMQ: A New Robust MQTT Broker for IoT

Amirhossein Sefati

April 14, 2022

Abstract

Nowadays, there are billions of devices around the world that are using Internet of Things protocols for their communication. There are plenty of protocols for doing that. Message Queuing Telemetry Transport (MQTT) is the most widely-deployed one among others. [1] However, MQTT has a problem that Eyhab Al-Masri et al called SPF. [1] MQTT has three elements in its protocol: publishers, subscribers, and brokers. The last one is like a server the publisher sends its message to it and then the subscriber will get that message from the broker. So, the problem is MQTT lies just on a single point, and it can trigger network failures when that single point is down, or it is not working properly. In this project, this problem will be solved using simulations and implementations of a new broker which we called DIMQ. The algorithm that is implemented is a novel chain-based idea that is more reliable, scalable, and feasible than other algorithms in the literature. This can affect the way that IoT (Internet of Things) platforms are using MQTT and brokers.

1 Introduction

1.1 Problem definition and motivation

As it can be seen from Figure 1, MQTT is working based on three elements:

- Publisher: A client which is sending messages,
- Subscriber: A client which is the receiver of the publisher's messages,
- Broker: A server that is like a mirror. Receives a message from the publisher and send back it to the subscriber.

In Figure 1, an imaginary scenario with 28 clients distributed in 6 states with one broker in Ontario is shown. This can explain the SPF problem easily. The broker is the single point of failure. If the broker goes down, The entire network will be out of service. MQTT protocol is designed to be centralized. But with the growth of IoT applications, being centralized can be a big problem. Figure 2 shows MQTT as the most used protocol in IoT big platforms such as Google IoT Core, IBM Watson IoT, Alibaba IoT, and Amazon AWS IoT Core. [1] Now, imagine a smart city with smart vehicles all connected through an IoT network that is using the MQTT protocol. This network is so big that all of the city is connected and being out of service even for a brief period can cause big economic loss (or even worse, it can be a danger to the healthcare system). But with the SPF problem, it will be a common thing that this network would be down because of broker failures. Right now, IoT platforms use multiple processors and multi-core ways for solving this issue by using load balancing methods such as using multiple instances of a docker container of the MQTT broker. Although these methods can enhance the reliability of the network and make it more reliable than the original MQTT brokers, they can't solve the main problem. Their way will be explained more and in detail in the "Previous Works" section. Therefore, by solving this problem, IoT MQTT-based networks will be more robust and scalable.

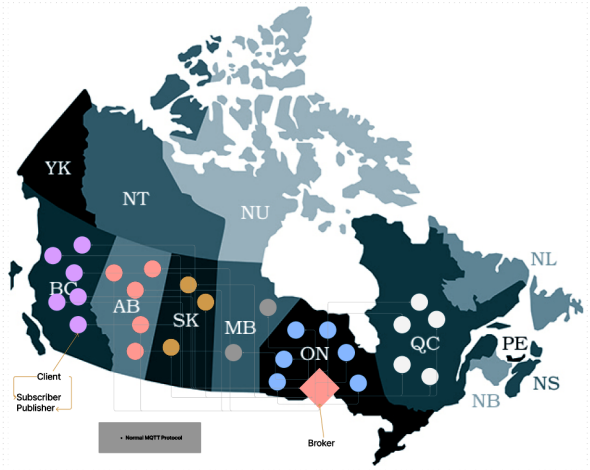

Figure 1: A Normal MQTT Network

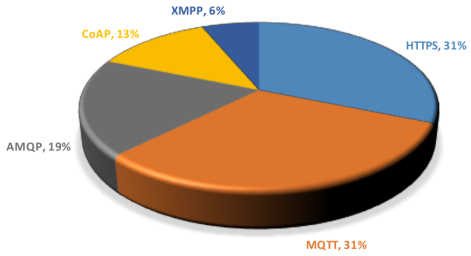

Figure 2: IoT Platform Support Distribution

1.2 Previous Works

Since MQTT was published generally in 2010, it is a relatively new area for investigation and research world. First, EYHAB AL-MASRI and his team investigated messaging protocols for IoT, and they found MQTT is the most interested one for the platforms.[1] Their article was published in 2020 and they found problems and issues for MQTT as well as some other protocols. Some issues they found important in the MQTT protocol are as follows:

- Topic names are often long,
- Doesn't support large payloads,
- Clients should support TCP.
- Broker can be a Single Point of Failure,

The first mentioned problem is a common thing when it comes to Messaging Protocols. Because we should have channels for our communications we should call them by a name, these names in MQTT are called Topic. They can be so long but with a good architecture for topics in the system, this problem can be solved. [6] The second problem they mentioned is not a big issue since IoT devices don't need to send large payloads such as multi-media. In fact, because of limitations of resources including power consumption, processor units, memory storage, etc. IoT devices are not able to send large payloads. They are not built for this purpose. The third problem matters too. MQTT is in the application layer of the network and it uses TCP protocol for its communication. So, clients (publishers and subscribers) should support TCP. But the main thing here is IoT devices are normally based on Raspberry Pi, Arduino, ARM, or AVR. So, they can handle TCP connections as well. The last problem that they mentioned in their work is what this project is investigating.

Through investigations of EYHAB AL-MASRI, the single point of failure problem has been mentioned multiple times and when it comes to attacks and security issues of MQTT in real-world, it has been mentioned that "A malicious client can then use this as a security exploit for possibly causing a

Table 1: Summary of Strengths and Weaknesses of IoT Protocols [1]

Criteria	HTTP	CoAP	MQTT	AMQP	XMPP	DDS
RESTful	high	high	none	none	none	none
Interoperability	low	low	low	moderate	high	high
Service Provisioning	low	low	low	moderate	moderate	low
Distributed Tracing	high	none	none	none	none	moderate
Scalable	low	low	moderate	moderate	moderate	high
Reliability	low	low	moderate	moderate	moderate	high
Security	low	low	low	moderate	high	moderate
Extensibility	low	low	low	moderate	high	moderate

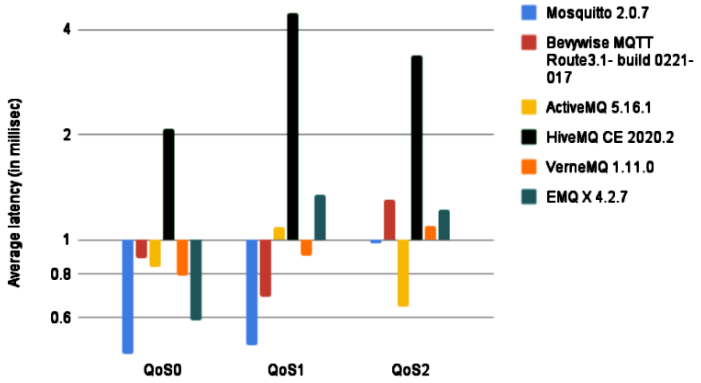

Figure 3: Average latency of all IoT brokers (in milliseconds) [2]

Denial of Service (DoS) attack." Also, they mentioned that the model of MQTT is "device to cloud", which is where the SPF issue appears.

Table 1 shows SPF issue affects the security, reliability, and scalability of MQTT in the following ways [1]:

- With DDoS attacks the broker will be down and the entire network will be out of service (the goal of the attacker),
- With the growth of the IoT network in several devices, the broker got too busy and it is statistically more common to be not responsible. Also, the maximum number of clients for one broker is 100,000 devices which is a limitation itself,
- The reliability of the network will be affected by the previously mentioned issues. So, we won't call that network reliable when it is probable for it to be out of service.

Another investigation was made in 2014 by Thangavel [5]. In his work, he conducted multiple experiments using a common middle-ware, to test MQTT and CoAP protocols, bandwidth consumption, and end-to-end delay. Their results showed that using CoAP messages showed higher delay and packet loss rates than using MQTT messages. But the main thing was both were not able to handle network or broker failures. In 2020, Biswajeeban Mishra and his team conducted a general investigation about brokers of MQTT protocol and their goal was to compare the performance of MQTT different brokers such as Mosquitto, EmqX, HiveMQ, ActiveMQ, etc. Figure 3 shows the latency comparison of all the brokers in the cloud evaluation environment. This table shows Mosquitto has a better performance than others when the quality of service is set to 1 or 2. Also, table 2 shows a bird's-eye view of the tested brokers, it can be seen that Mosquitto is free to use and it is written in the C programming language. Because of these two reasons and considering that EMQ X, HiveMQ, and ActiveMQ are not open-source (they have free and pro versions), I will use Mosquitto as the base broker and I will try to enhance the performance of this broker throughout the project.

In 2020, Edoardo Longo and his team at the University of Cambridge conducted research to find a solution to the SPF problem in the MQTT protocol. The Spanning Tree Protocol is presented in the paper. [4] It is a distributed protocol that creates a logical spanning tree over a meshed network of

Table 2: A bird's-eye view of the tested brokers [2]

MQTT Brokers	Mosquitto	Bevywise MQTT Route	ActiveMQ	HiveMQ CE	VerneMQ	EMQ X
OpenSource	Yes	No	Yes	Yes	Yes	Yes
Written in	C	C, Python	Java	Java	Erlang	Erlang
MQTT Version	3.1, 5.0	3.1, 5.0	3.1	3.1, 5.0	3.1, 5.0	3.1
QoS Support	0, 1, 2	0, 1, 2	0, 1, 2	0, 1, 2	0, 1, 2	0, 1, 2

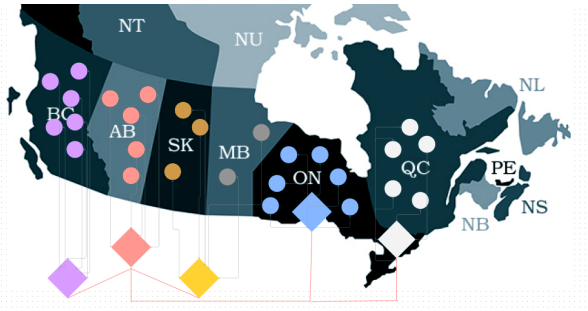

Figure 4: MQTT-ST: a Spanning Tree Protocol for Distributed MQTT Brokers

layer 2 switches. Such a spanning tree is obtained by picking a root switch and blocking some of the output ports of the other switches: blocked ports do not send data frames, thus avoiding broadcast storms. It has been implemented by the team. However, it was not working properly. Also, it has nothing to do with clients of that "down" broker. It means Edoardo Longo investigates a tree of brokers. Figure 4 shows what they implemented. As they mentioned: "In our previous position paper, we analyzed the main research challenges and possible solutions to scale up a pub/sub architecture for upcoming 5G networks, and we presented our view on system design and optimization.", In the previous paper from them, they recognize SPF problem of MQTT. Their MQTT-ST broker has the following features: [4]

- Connection phase: First, to allow bidirectional communication, the broker transmits back a modified CONNECT message to the originating node, using the standard MQTT port 1883. The broker stores the IP addresses of all directly connected brokers in a local table, which is used to keep track of the state of each connection marked as root. Finally, the broker sets itself as the root and starts transmitting signaling messages to all connected brokers. [4]
- Signalling phase: Standard MQTT specifies a Keep-Alive parameter, which defines the maximum time interval permitted to elapse after the last client transmission. In case the timer expires, the broker closes the connection with the client. [4]
- Root selection: The root broker plays a crucial role in the broker tree, as it is the relay node for all traffic and it is, therefore, subject to an increased computational load. Indeed, selecting a broker with poor or overloaded resources may result in poor overall performance. In STP the root is selected based only on its identifier, which does not suit well the scenario under consideration. In MQTT-ST, instead, the root broker is selected according to the capability value C. It is defined as: $C = \alpha L + \beta M$. In the mentioned formula, L is the broker CPU speed, R is the amount of RAM, and α and β are changeable conversion parameters. [4]
- Path computation: In STP, each node selects the best path to the root according to bandwidth-related criteria, to avoid the use of reduced capacity links in the tree which may slow down an entire network. For MQTT-ST we observe that latency, rather than bandwidth, plays a critical role. [4]

Table 3 shows the similarities and differences between this project (DIMQ), Edoardo Longo's work (MQTT-ST), and a normal Mosquitto broker. We consider "Locality" as the measurement for equal distribution of clients among brokers. In the results section, we can see that locality plays a big role in the latency of the network.

Table 3: Differences and Similarities Between this project and previous works

Attribute	DIMQ	MQTT-ST	Mosquitto
Base Broker	Mosquitto	Mosquitto	-
MQTT Version	3.1 and 5	3.1 and 5	3.1 and 5
Problem to solve	SPF	SPF	-
Main Methodology	Chain-based model	Tree-base model	-
Implementation	Yes	Yes	Yes
Client Importance	High	Moderate	Low
Latency	5ms	5ms	4.5ms
Reliability	High	High	Moderate
Scalability	High	Moderate	Low
Structure	Distributed	Distributed	Centralized
%100 Locality*	Normal	Poor	Normal
Additional Features	Client and Broker Feedback	Semi-Broker Feedback	-
OS Support	Linux	Linux	Linux, Windows, Mac
Year Published	2021	2020	2015

1.3 Key Contributions

In this project, we bring a novel idea for tackling the problem of a Single Point of Failure. Table 3 shows many differences and similarities between our work and previous works. The key contribution is with this new algorithm I got the same latency as the tree-based model while it is better in fields of client importance and scalability. In this algorithm, brokers will connect using a chain that can store the state of the entire network. In the previous work (MQTT-ST), if one of the brokers failed to respond, the entire structure should get updated, and the "Selection of Root" algorithm of their work restarts. This means with just little changes in the elements of the network, everything has to change. But in a chain-based model that I am proposing and implementing, failing one broker doesn't change many things in the chain (This will be explained more in the discussion section). Also, the MQTT-ST broker has a problem that if all of the clients are connected to the same broker among the tree, the performance will be very poor. They called this situation a %100 Locality problem. However, DIMQ tackles this problem by giving feedback to the client. The feedback includes the average latency of messages (RTT), IPs, and ports of other brokers in the chain and their load capacity. Whenever clients have enough data about the latency of their messages, they can switch to other brokers to get better latency and better performance.

1.4 Methodology

In this project, a new algorithm has been proposed to tackle the SPF problem of the MQTT protocol. The algorithm has 5 main situations.

- Initialize Chain: At first, the main broker which is at the start of the chain should start in the public mode. This could happen using a configuration file (i.e. dimq.conf). The first broker in the chain is just a broker and it is not connected to any other brokers. Next new broker will connect to this broker to make a chain of brokers. But the first broker never connects to any brokers. This helps the network to prevent infinite loops.

- Adding a new broker: If any other broker wants to join the chain, it should have IPs and ports of all of the other brokers in the chain concerning the time they joined the chain. Information is stored in the specified configuration file using MQTT detail syntax. For example, if we are adding n_th broker to a chain which contains $n-1$ brokers, it should have all the other brokers with the syntax of: addresses $IP_{n-1} : Port_{n-1}$ $IP_{n-2} : Port_{n-2}$ $IP_{n-2} : Port_{n-2}$ ... $IP_1 : Port_1$. This should be in the configuration file. There is an algorithm for switching between brokers which are called "Round Robin". This specifies how the priority of broker connections. If the round-robin algorithm is turned off, brokers will always prefer connecting to the first IP in the configuration file. But if this algorithm is turned on, brokers will always prefer the now-alive connection and won't switch to the first one. Theoretically, the last configuration is better because it causes much fewer changes in the network structure and its chain of brokers.

- Failure of a broker: When a broker fails to respond, it means it is down (Itself or the link to which it is connecting is corrupted), the broker which was connecting to the down one will switch to a new broker which down-broker was connected to. For example, imagine m^{th} broker is connected to the $m + 1^{th}$ broker and $m - 1^{th}$ broker is connected to the m^{th}. Now, if the broker number m failed to respond, broker number $m - 1$ will connect to broker number $m + 1$. By doing this, the entire network will remain working properly and the only change here is changing one connection. Another thing here is that the clients which were connected to the m^{th} broker, shouldn't be lost. In DIMQ, brokers publish their IP and Ports in a special internal system topic: $Brokers/IP. When they are connected to the network, the value of the message is just their IP of them, but if they are disconnected, the broadcast will occur and the value of the message will be "/off" which shows that the broker is not in the chain anymore. With the extra information provided by DIMQ brokers, clients of the down broker can switch to the best performance broker. This is like an optimization problem that should be solved in a real-time environment by clients. Solving optimization problems in real-time needs many resources and will increase power consumption. So, DIMQ clients should be able to turn this feature on/off depending on the situation. When this feature is off (default mode), they just connect to the next close available broker using IP Geolocation service. Also, when a broker is down and up again, it should connect to the last broker. It helps the entire network in two important aspects:

  1. Improve load balancing when a broker called α is down, probably, α is not powerful enough to maintain the activities of the network, so it is better for the whole network if α is added to the last position in the chain so no broker is connected to it and the load on this broker will be less than others.
  2. Decrease the number of changes when a broker gets down and up again in the network structure. For example, if a network has n brokers in its chain and m^{th} broker is down and up again, if it connects to the last broker, only one change occurs. But if it connects to the $m + 1^{th}$ broker like before, the $m - 1^{th}$ broker should be disconnected from the $m + 1^{th}$ and connects again to the m^{th}, with this strategy 2 changes occur. So, we prefer the first strategy.

- The last broker in the chain: The last broker on the chain is the only one that is connected to a broker but no broker is connected to it. The chain is not closed, so it doesn't have any loops. This was necessary to prevent making infinite loops where the message will be published more and more with no stops.

- Link failures: There is a situation in which a link between two brokers is broken. Imagine the link between m^{th} and $m + 1^{th}$ broker is broken, in this situation, the m^{th} broker consider this as the $m + 1^{th}$ broker is down, so it tries to connect to $m + 2^{th}$ broker. As a result, the chain is like a tree with a depth of 2. So, link failures are considered broker failures and will generate tree-based models instead of chain-based models.

In the next section, first, each of these 5 situations is modeled and explained by an example scenario. Then, the result of the final algorithm is shown. In the end, there is a comparison between the results of this project and the results available in the literature. After all, a client is shown who uses DIMQ brokers to communicate with other clients. The client is an Android application that is implemented in a way messengers work. Finally, it is used to evaluate the DIMQ broker and its algorithm.

2 Distributed Intelligent MQ (DIMQ)

2.1 Modeling the behavior

Algorithms 1 to 4 show how DIMQ brokers work in different scenarios. All of these scenarios are implemented in the DIMQ source code and it is working properly. The code is available on my GitHub page: See the source code.
The code is a fork of Mosquitto broker which is fully open-source written in C programming language, the difference is the additional algorithms for being reliably distributed. There are multiple examples of how to write *dimq.conf* files in the before mentioned GitHub directory. One example of writing this file is as follows:

#Bridge you to last (This is like a comment in .conf files)
connection B03To02
addresses 74.208.35.55:1883 135.55.42.67:1883 90.223.34.56:1883
notification-topic $Brokers/me$
topic # out 0
topic # in 0
round-robin 1

Algorithm 1 DIMQ brokers algorithm: initializing the chain

Require: $numberOfBrokers = 0$
Ensure: $numberOfClients = 0$
  $numberOfBrokers \leftarrow 1$
  if $CONNECT$ have ACK **then**
    Publish IP in $\$Brokers/IP$ topic with $retain = 1$ ▷ For 2^{nd} broker.
    $RoundRobin \leftarrow 1$
    Accept connections from clients
  else if $Error$ **then**
    $numberOfBrokers \leftarrow 0$
    System.Restart()
  end if

Algorithm 2 DIMQ brokers algorithm: adding a new broker

Require: $numberOfBrokers \geq 1$
  $numberOfBrokers$ += 1
  Run from *dimq.conf* file.
  Connect to the last IP in the topic of $\$Brokers/IP$. ▷ Develop the chain.
  if $CONNECT$ have ACK **then**
    Publish IP in $\$Brokers/IP$ topic with $retain = 1$ ▷ For the next new broker
    $RoundRobin \leftarrow 1$
    Accept connections from clients
  else if $ConnectionLost$ **then**
    Publish IP/Off in $\$Brokers/IP$ topic with $retain = 1$ ▷ Feedback to clients
    $numberOfBrokers$ -= 1
    System.Restart()
  end if

Algorithm 3 DIMQ brokers algorithm: failure of a broker

Require: $numberOfBrokers > 1$
$numberOfBrokers$ += 1
For Failed Broker (m^{th}):
- Publish IP/Off in $\$Brokers/IP$ topic with $retain = 1$ ▷ Feedback to clients
- $numberOfBrokers$ -= 1
- System.Restart()
For $m - 1^{th}$ Broker:
- Connect to last IP in $\$Brokers/IP$ topic with no value ending $/Off$
- Get ACK from $m + 1^{th}$ and run $Algorithm2$ if necessary.
For Clients of m^{th} broker: ▷ Use KNN instead or just connect blindly
- Search from values in $\$Brokers/IP$ topic for the most optimized broker to connect.
- Use IP Geo-Location Service to find most close broker.
- Delete m^{th} broker from table.

Algorithm 4 DIMQ brokers algorithm: failure of a link

Require: $numberOfBrokers > 1$
Run $Alogorithm3$ for the broker which is topper in the stack.
if Link was between 1^{st} broker and 2^{nd} broker **then**
1^{st} broker ← last broker ▷ When the link is up again
2^{nd} broker ← 1^{st} broker.
end if

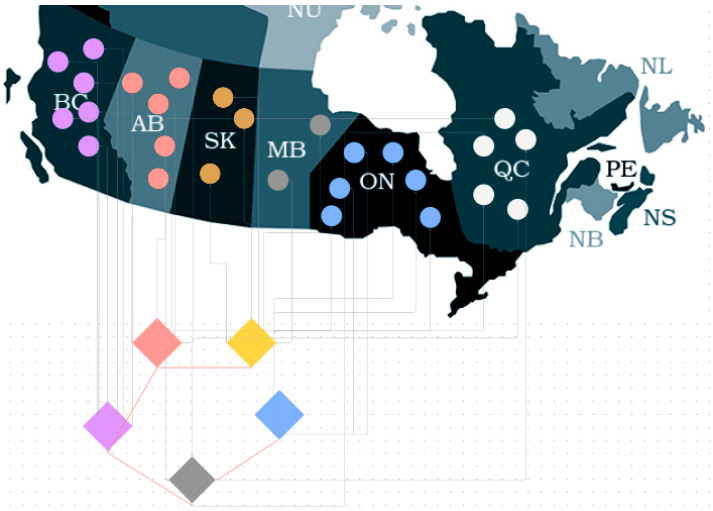

Figure 5: DIMQ: A Chain-based Distributed Model

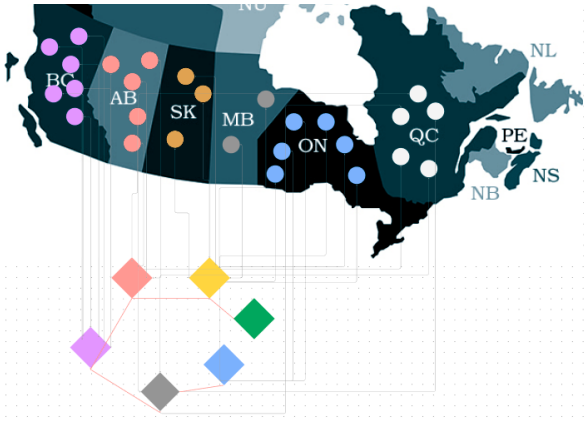

Figure 6: DIMQ: Adding a new broker

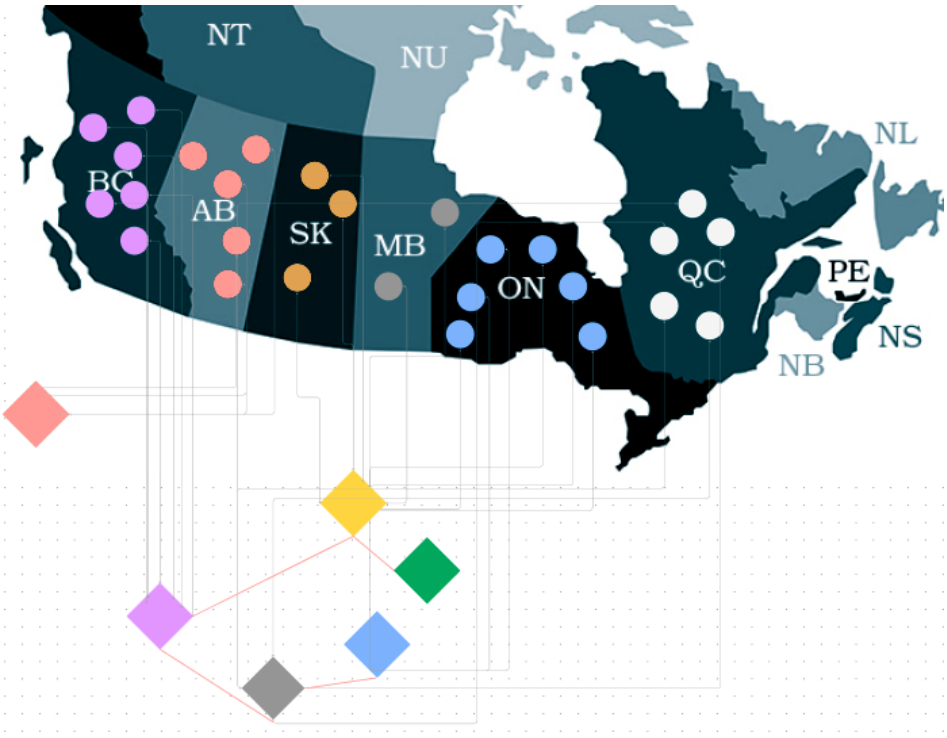

Figure 7: DIMQ: Failure of a broker - Before Re-connection of Clients

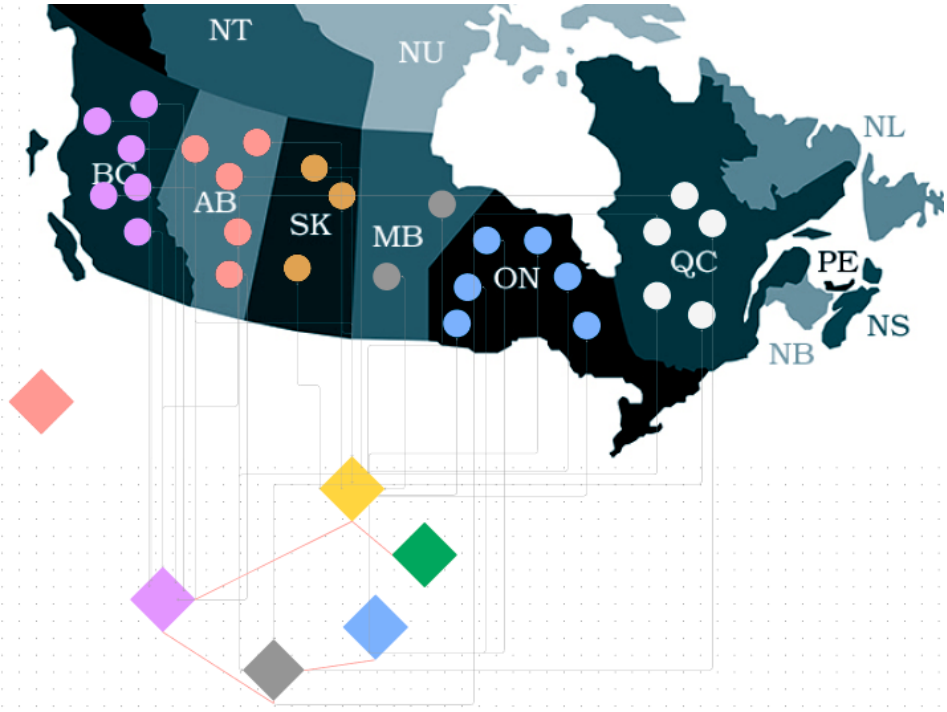

Figure 8: DIMQ: Failure of a broker - After Re-connection of Clients

2.2 Evaluation and Results

2.2.1 Results of DIMQ Broker

In this section, the DIMQ brokers had been tested in many different configurations. We use four different variables for the network: 1- Number of brokers which are connected using the bridge, we evaluate the system for 1 to 10 brokers. 2- Number of clients which each is connected to a broker randomly from the broker's list in our simulation code. We evaluate the system for 1 to 200 clients. 3- Number of publishes per client which is the total number of packets being sent from each client. We evaluate the system for 1 to 15200 publishes per client. 4- Number of subscriptions per client which is the total number of MQTT channels that each client subscribed to. We evaluate for 1 to 100000. However, we end up realizing that the number of subscriptions is not affecting any of the two results that we were looking for. So, we decide to just evaluate for 1 to 2000 to speed up the process of evaluation. Google, in its article about Pub/Sub messaging services, described three following factors as the most important system properties to measure.

- Latency: it denotes the time the service takes to acknowledge a sent message or the time the service takes to send a published message to its subscriber. Latency can also be defined as the time taken by a messaging service to send a message from the publisher to the subscriber. [3]
- Scalability: Scalability usually refers to the ability to scale up with the increase in load. A robust scalable service can handle the increased load without an observable change in latency or availability. [3]
- Availability: Systems can fail. Sometimes a sudden increase in load results in resource shortage and thus causes a system failure. We consider load as the number of publishers and subscribers. [3]

We measured latency as the time that it takes for a message to be published from the publisher and received by the subscriber as well as the success rate. Also, we measured the locality of clients as [4] believed locality plays the most important role in the system performance. We define locality with the definition of standard deviation. So, if we have 10 brokers and 100 clients in our simulation process, at the locality of zero, we should have 10 clients for each broker. But higher locality shows a non-balanced client distribution over brokers. Results are generated from our simulation program. To eliminate the impact of network latency, we use a local server for testing the system. The simulation was on a local server with 16 GB of RAM, 7 cores of CPU (Intel 8th edition), and 512 GB PCIe SSD. At maximum, we used 10 brokers (with the same local IPs but with different ports), 200 clients publishing 15200 packets in each iteration, and subscribing into 2000 channels at the creation phase. Table 10 (In the appendix section) shows the results of DIMQ with different configurations. Here we shortly mention some observations which are resulted from the evaluation:

- When it comes to latency, the number of connections/clients is more important than the number of publishes or subscriptions. Table 4 shows this observation in the behavior of our system.
- Table 5 shows that number of publishes affects the performance of the system by increasing the latency. But this effect is not as important as the number of connections/clients.
- Distributing the same number of clients through a network of interconnected brokers decreases the latency of the system. It can be seen in Table 6. So, DIMQ (or the previous work, MQTT-ST) has a better performance in the same environment compared to a basic MQTT broker which doesn't support interconnected brokers.
- As they said in [4], the locality has the most influence on the performance of the system. Table 7 shows many cases of this fact. As we can see, with the same conditions, just because the random creation of clients caused them to be distributed unbalanced, the latency increased by a high value. As the two last rows in the table offered, even with more clients and more number of publishes, if the locality is managed better, latency will be handled too.

So, we can see the locality problem in DIMQ. It can validate what [4] claimed before. Therefore, we should change how the network of DIMQ brokers should work together in future works.

Table 4: Number of Clients' Effect

# Brokers	# Clients	# Publishes	# Subscribes	Locality	Latency (seconds)	Success Rate
1	50	3800	50	0.0	0.0108573	1.0
1	100	3800	50	0.0	0.0158573	1.0

Table 5: Number of Publishes Effect

# Brokers	# Clients	# Publishes	# Subscribes	Locality	Latency (seconds)	Success Rate
1	100	3800	50	0.0	0.0158573	1.0
1	100	5100	100	0.0	0.0172135	0.9101

Table 6: Number of Brokers Effect

# Brokers	# Clients	# Publishes	# Subscribes	Locality	Latency (seconds)	Success Rate
1	200	15200	2000	0.0	0.0229697	0.9869
10	200	15200	2000	80.0	0.0084695	1.0

Table 7: Locality Effect

# Brokers	# Clients	# Publishes	# Subscribes	Locality	Latency (seconds)	Success Rate
2	50	3800	50	128.0	0.0197734	0.9913
2	50	3800	50	2.0	0.0109484	1.0
2	150	7650	1500	0.0	0.0099347	1.0
2	150	7650	1500	50.0	0.0239099	0.9388
3	50	3800	50	44.666	0.0209715	1.0
3	50	3800	50	0.6666	0.0123863	1.0
3	150	7650	1500	326.0	0.0232080	0.8352
3	150	3900	1500	62.0	0.0130887	0.8830
3	150	11400	1500	26.0	0.0090986	1.0
3	200	5200	2000	24.666	0.0228147	0.9536
3	200	5200	2000	200.66	0.0327350	0.9347
5	100	7200	100	10.0	0.0138931	0.9961
5	100	7200	100	106.0	0.0170100	0.9992
10	150	3900	150	136.0	0.0118530	1.0
10	200	15200	2000	80.0	0.0084695	1.0

However, in the current version of DIMQ, the network can handle Single Point of Failure (SPF) successfully. It is evaluated in the next subsection.

2.2.2 Results of DIMQ Client

DIMQ has another advantage compared to normal Mosquitto and also MQTT-ST: Importance of clients. It uses feedback signals to clients to inform them whenever a broker fails. In Algorithm 3 it has been mentioned that when a broker fails, clients which were connected to that broker can switch to other brokers using IPs that are in the topic of $Broker/IP$. As the clients are the main focus of this project, the main development takes place on the client side. So, as an experiment, an React Native application has been developed during the semester. Users can sign-up for the application and start conversations with other users. Some of the features of this application are the followings:

- Set a username for the account,
- Sign-in using that username (It will be recognized as a client in the network by that username and it is not changeable),
- Send/Receive Text-based messages,
- Auto Reconnect to available brokers,
- Separated private rooms for users,
- See available users in the network. (It is a feature with DIMQ brokers: No matter to what broker they are connected, they can find each other)

Table 8 shows the conditions of 2 devices on which $DIMQ\ Client$ were tested on them. Each client then connects to the network and publishes a message with the value of its username and topic of $Users/Username$. This message has a retain flag set to 1. It means that this message will be stored in the entire network the same way the IPs are stored. So, in the DIMQ network, we have a track of all the clients. Additionally, normal messages will be sent with the enabled "clean-session" option. This means that the message will be stored in the network until the client receives it. For example, client A sends a message with the value of M to the client B. But client B is offline and it is not available to receive the message, DIMQ broker will store this message in its distribution network. When client B is online again, the DIMQ broker will pass message M to this client and then remove this message from the entire network. Client B can save that M message for future use. The implemented $DIMQ\ Client$ in this project saves all the private messages in its memory. So, all of the communications between clients are stored on their own devices and not in any brokers (servers). It should be mentioned that this is a feature that basic MQTT supports as well.

Figure 11 shows $DIMQ\ Client$ running in the virtual Android simulator. It is connected to DIMQ local broker. The Code of this client is written using JavaScript programming language and the source is available on our GitHub page in the DIMQ_M repository. (See the repository) This uses a JS library called "react-native-mqtt" to be able to communicate using the MQTT protocol. The installation guide is available on the official page of the react-native community. The testing scenario is as follows. Also, it is shown in Figure 9.

1. There are 2 devices with the configuration of table 5 in the network,
2. There are 3 DIMQ brokers,
3. One of the clients is connected to the public DIMQ broker (Pixel 5),
4. Other client is connected to the third broker (Galaxy S8),
5. Suddenly broker number 2 is down (we just disconnected broker number 2 using terminal commands)
6. Observe whether DIMQ brokers can handle connection failure. As a result of failing broker number 2, broker number 3 which was connected to the second broker, connects to the public broker immediately.
7. Now, suddenly broker number 3 is down,

8. Observe whether *DIMQ Client* which is running on Galaxy S8 can handle broker failure. As a result of the failure of broker number 3, the client connects to the public broker. Because it has no connection to broker numbers 2 and 3.

At the end of the testing, all of the components performed well and the entire network remain working properly. Two clients were able to continue their communication through the DIMQ network without any problems in sending or receiving messages. It doesn't even need any reloading of the application or page. Figure 11 shows how MQTT X software help us through the project since we couldn't run 2 simultaneous Android simulators in our system. As figure 9 shows, there should be a special architecture for designing the MQTT topics in any system, like the Amazon provided in the article "Designing MQTT Topics for AWS IoT Core" [6]. For this project, the architecture of the communication in *DIMQ Client* for sending and receiving messages is as follows:

1. There are two public topics: $\$Brokers/IP$ for information on brokers in the chain and $Users/USERNAME$ for information on clients in the network.

2. Any client is subscribing to the following topics:

   - $\$Brokers/\#$
   - $Users/\#$
   - $USERNAME/\#$: For example, if the username of the client is Amirhossein, it subscribes to $Amirhossein/\#$ topic as soon as it is connected to the network.

3. Publisher always publishes its message in the following format of topic:
   $USERNAME\_Of\_Receiver/USERNAME\_Of\_Sender$
   For example, if *Amirhossein* wants to send a message to *David*, the topic should exactly be like "David/Amirhossein". And vice versa if *David* want to send a message to *Amirhossein*

Table 8: Configuration of each mobile phone as DIMQ client in the test

#	Real/Virtual	IP	Port	Model	Android
1	Virtual	192.168.0.96	1883	Google Pixel 5	Version 9
2	Real	192.168.0.96	1884	Samsung Galaxy S8	Version 8

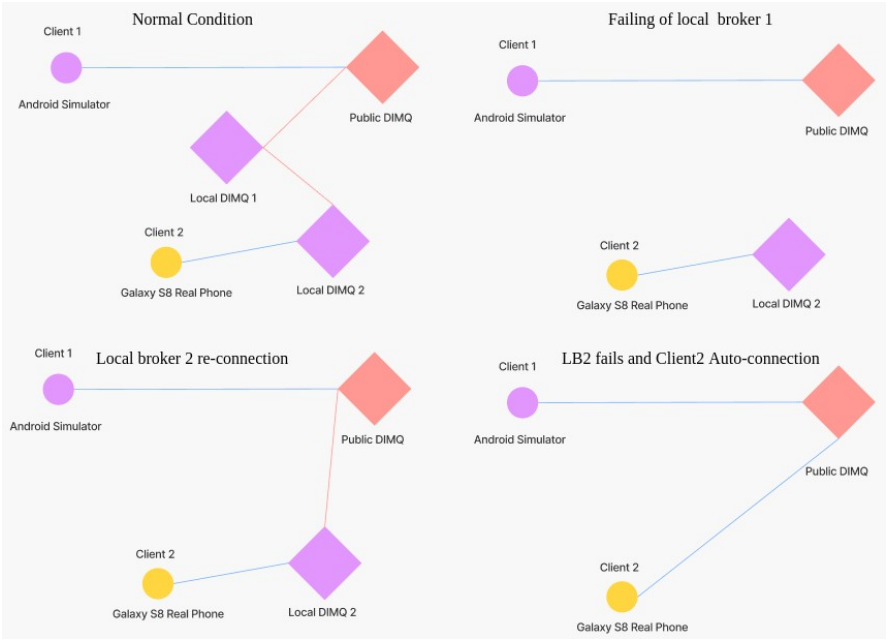

Figure 9: Testing of Clients and Brokers in DIMQ Network

3 Conclusion

The results of this project show that DIMQ can make a more reliable and scalable network on the MQTT protocol. It means that DIMQ can be a great option if the network includes many clients (publishers and subscribers) and it needs to be always accessible to them. DIMQ has the same latency as MQTT-ST while it is more reliable and values the needs of clients in every situation. The broker, publisher, and subscriber are implemented and tested in the real network environment. So they can be used in real-world projects as well as research. DIMQ uses feedback to inform clients when a broker fails, it helps the network to continue its work without losing any clients. Observations from the evaluation section show that the locality problem is affecting latency and DIMQ has the ability to solve this in the future. Also, we prioritize the variables of an MQTT-based network in the evaluation section. We programmed a simulator for testing any MQTT-based network and we tested our DIMQ with that program.

3.1 Limitations and Advantages

Through this project, we experienced the following limitations:

- We couldn't run the simulation for more than 200 clients and more than 15200 publishes, the operating system of our local server prevent us from doing that with the "too many open connections" error.
- We were limited to testing this broker on Linux 18.4 since we didn't have access to other operating systems.
- Since the time was limited, we didn't have enough time to perform more tests and get the most confidential results.

3.2 Recommended Future Works

During the development of this project, we noted some useful approaches to continue DIMQ to gain better latency and more reliable data transfer. Following is the list of the works that can be done in the future:

- As section 2.2.2 shows and [4] claimed before if the clients are not equally distributed through the brokers, latency will be increased. We can use Machine Learning algorithms to decrease the latency by decreasing the locality of the entire network. For an instance, KNN can help us to identify locality issues and assign devices to desired brokers in order to balance the network. For doing that, the current version of DIMQ has all the information which is needed for the process. We just need to improve how DIMQ clients should decide whenever they are created.
- The DIMQ network should be able to automatically update dimq.conf in the source. It can help to have a general, worldwide use of this broker which can be open for anyone to contribute by running a DIMQ broker.

References

[1] E. Al-Masri et al., "Investigating Messaging Protocols for the Internet of Things (IoT)," in IEEE Access, vol. 8, pp. 94880-94911, 2020, doi: 10.1109/ACCESS.2020.2993363.

[2] Mishra B, Mishra B, Kertesz A. Stress-Testing MQTT Brokers: A Comparative Analysis of Performance Measurements. Energies. 2021; 14(18):5817. https://doi.org/10.3390/en14185817

[3] Google Cloud. (n.d.). Pub/Sub: A Google-Scale Messaging Service. Google. Retrieved February 10, 2021, from https://cloud.google.com/pubsub/architecture

[4] E. Longo, A. E. C. Redondi, M. Cesana, A. Arcia-Moret, and P. Manzoni, "MQTT-ST: a Spanning Tree Protocol for Distributed MQTT Brokers," ICC 2020 - 2020 IEEE International Conference on Communications (ICC), 2020, pp. 1-6, doi: 10.1109/ICC40277.2020.9149046.

[5] D. Thangavel, X. Ma, A. Valera, H. -X. Tan and C. K. -Y. Tan, "Performance evaluation of MQTT and CoAP via a common middleware," 2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014, pp. 1-6, doi: 10.1109/ISSNIP.2014.6827678.

[6] Amazon. (2021, December 8). Designing MQTT Topics for AWS IoT Core. Amazon Web Services. https://docs.aws.amazon.com/whitepapers/latest/designing-mqtt-topics-aws-iot-core/designing-mqtt-topics-aws-iot-core.html

4 Appendix

Table 9: Configuration of each broker in our test

#	Public/Local	IP	Port
1	Local	192.168.0.96	1883
2	Local	192.168.0.96	1884
3	Local	192.168.0.96	1885
4	Local	192.168.0.96	1886
5	Local	192.168.0.96	1887
6	Local	192.168.0.96	1888
7	Local	192.168.0.96	1889
8	Local	192.168.0.96	1890
9	Local	192.168.0.96	1891
10	Local	192.168.0.96	1892

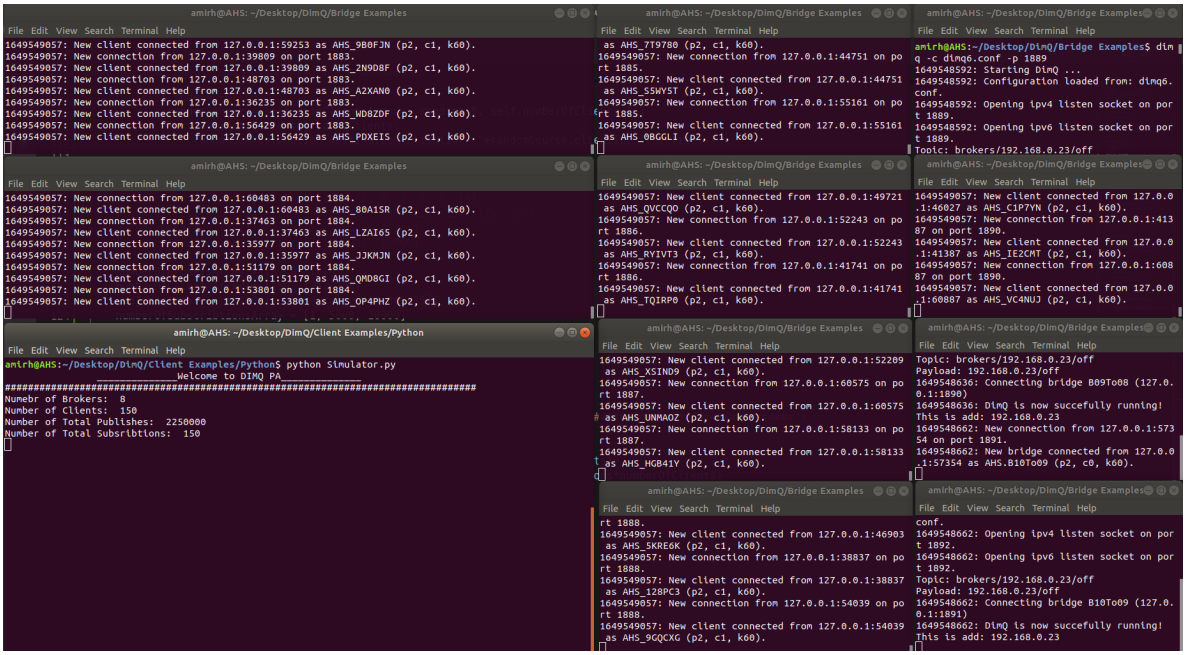

Figure 10: The testing phase of our project - 8/10 active brokers - 150 clients

Table 10: Results of our test

# Brokers	# Clients	# Publishes	# Subscribes	Locality	Latency (seconds)	Success Rate
1	1	1	1	0.0	0.0005373	1.0
1	50	3800	50	0.0	0.0108573	1.0
1	100	3800	50	0.0	0.0158573	1.0
1	100	5100	100	0.0	0.0172135	0.9971
1	150	7650	150	0.0	0.0174292	0.9908
1	200	15200	2000	0.0	0.0229697	0.9869
2	1	1	1	0.0	0.0009293	1.0
2	50	3800	50	128.0	0.0197734	0.9913
2	50	3800	50	2.0	0.0109484	1.0
2	100	7200	1000	0.0	0.0131289	1.0
2	150	7650	1500	0.0	0.0099347	1.0
2	150	7650	1500	50.0	0.0239099	0.9388
2	200	15200	2000	8.0	0.0189312	0.9646
3	1	1	1	0.0	0.0010458	1.0
3	50	3800	50	44.666	0.0209715	1.0
3	50	3800	50	0.6666	0.0123863	1.0
3	100	7200	1000	20.666	0.0130156	1.0
3	150	7650	1500	326.0	0.0232080	0.8352
3	150	3900	1500	62.0	0.0130887	0.8830
3	150	11400	1500	26.0	0.0090986	1.0
3	200	5200	2000	24.666	0.0228147	0.9536
3	200	5200	2000	200.66	0.0327350	0.9347
5	1	1	1	0.0	0.0009758	1.0
5	50	3800	50	54.0	0.0134243	0.9952
5	100	7200	100	10.0	0.0138931	0.9961
5	100	7200	100	106.0	0.0170100	0.9992
5	150	11000	150	34.0	0.0121746	1.0
5	200	15200	200	150.0	0.0192993	0.9659
10	1	1	1	0.0	0.0008327	1.0
10	1	76	10	0.0	0.0008282	1.0
10	150	150	1500	44.0	0.0085198	1.0
10	150	3900	150	136.0	0.0118530	1.0
10	200	15200	2000	80.0	0.0084695	1.0

Figure 11: Android Simulation of *DIMQ Client*: Main page

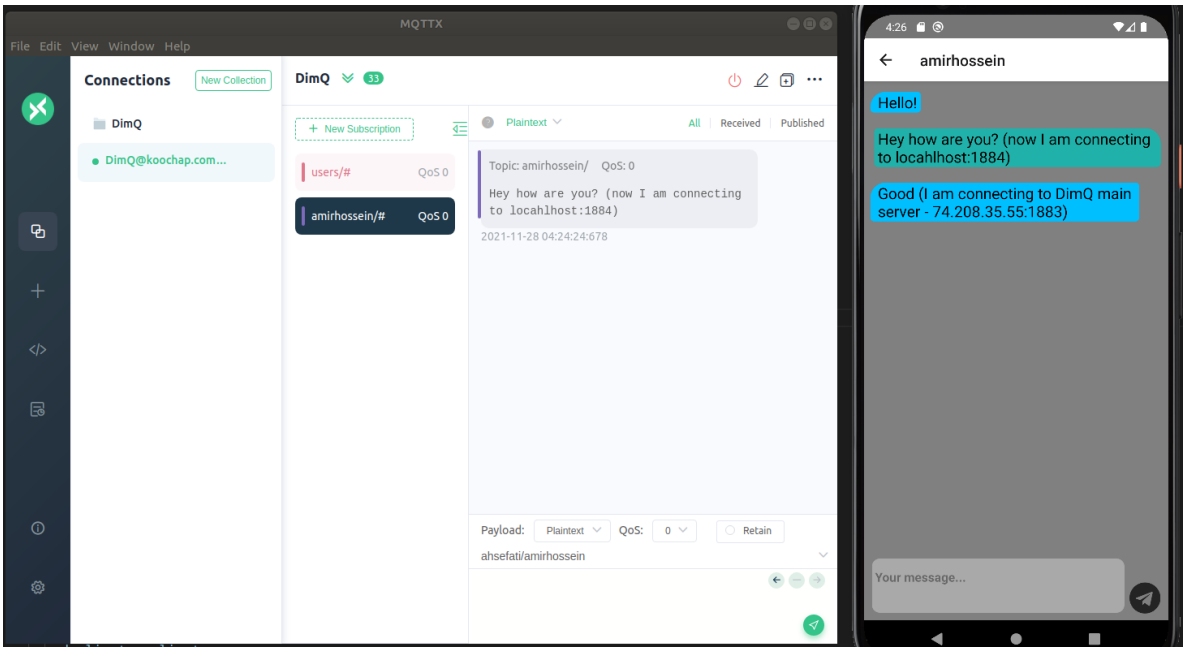

Figure 12: Android Simulation of *DIMQ Client*: Chat page