Fault Tolerant File-locking Service

Gupta
School of computing, informatics,
decision systems engineering
Arizona State University
Tempe, USA
agupt223@asu.edu

***Abstract*— This paper deals with the designing and implementation of a fault tolerant file locking system. It provides insights on the MQTT protocol and how it is used in this system.**

Keywords—server, client, MQTT Broker, MQTT Client, Leader Server, Backup Server, Publish, Subscribe.

I. Introduction

In the world of computers, reading and writing files is an elementary job. Whenever a computer is used with an application, if there is any information that is to be saved, it is done by writing to files. We do this all the time. When we write a report or make a presentation, we always write to files. When we are researching, we are reading from files. But what if we try to open two instances of the same file? Further, what if we try to simultaneously write to the two different instances of this file, such that the two instances of this file conflict? These things should not happen! And normally this is taken care of by the operating system. But the problem becomes more complicated if we include a server in the system. There are files on the server which have to be operated on by the clients. These files are being shared amongst all the clients. Any client can ask for any file to be read, written, opened or closed. While one client is operating on one of the files, it is imperative that no other client should be able to operate on that file; if the file is opened in write or read/write mode. To ensure this we need to make a file locking service on the server end which can keep a track of all the files and clients. **In this paper, we first go through the architecture of the system, then through a little deeper work flow challenges, experimental setup and future work.**

II. Related Work

RAFT is a system which tries to implement a fault tolerance by replicating servers and holding leader elections after some time.

There have been many attempts at making a 100 percent accurate and efficient fault tolerance, but many of them have flaws. They are usually specifically tailored to suit a particular application or environment.

III. Architecture

In a computer based communication, structure i.e., architecture and configuration of components, as well as protocol of communication are which decide the quality and reliability of communication. There is a server and many clients. The server caters to clients' requests. The clients request the server for accessing, reading from and writing to files which are there on the server. This is done via UDP protocol. Thus, it is possible that a client makes a request and this request never reaches the server. Thus if the client does not receives a response within a specified amount of time then it resends the request to the server. Similarly it is possible that the server sends the response but it gets lost on the way back and never makes it to the client. In this case also the client times out and resends the request. But as in this case the request was already catered to, the server only forwards the pre-recorded response to the client and does not performs the request again. This system ensures that even if there is some fault in the UDP communication, the system still holds together. But now a question arises that what if the server itself dies? In this case the client will forever keep sending the request and never get any response. To overcome this problem, we go for a fault tolerant system. A system similar to raft but not exactly the same.

The structure of the system is of utmost importance. There are four main players in the system. The first is the client, then the Leader server, then the backup server(s) and finally the MQTT broker. The role of the client is simply to request the server for reading, writing and closing files. The leader server accepts this request and caters it. The client talks only to the leader server. This communication between the client and the server, again, is always using UDP protocol. The aim, now, is to make this system into a fault tolerant one. So we now introduce the backup servers in the play. The role of these backup servers, at large, is to keep catering requests in the background alongside the leader server (of-course not sending the response) without making its/their presence felt to the client. Another one of its role is to guide the client to the real leader server, should a client accidentally directly communicate with it. Lastly, what the MQTT broker is, can only be understood by comprehending the MQTT protocol.

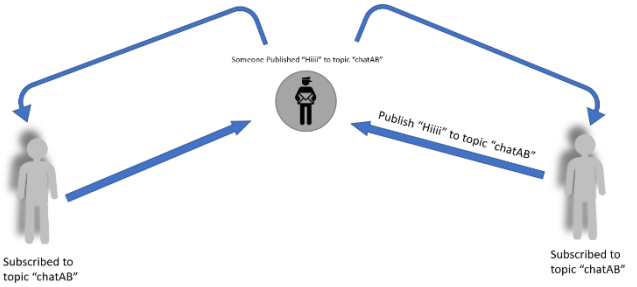

Fig 1. Example of the MQTT Protocol

To understand the MQTT protocol we need to know its architecture. There are two players in this system. First the MQTT Broker and the second, the MQTT Client. There is obviously only one broker and there are a number of possible clients. The MQTT protocol is a simple communication

protocol between clients, which eliminates the need of a server. If there are 5 MQTT Clients and one client wants to send a message to any other client then they simply have to inform the broker about this. How this happens will become clearer by looking a step further into the working of this protocol. This protocol is different from normal communication protocols like TCP and UDP protocols, thus understanding and appreciating this may take time; nevertheless upon completion of this paper it will become clearer. The MQTT Clients can tell the broker that they are interested in knowing about some specific topic. Now when any MQTT Client drops a message on the MQTT broker about this particular topic, the MQTT Broker simply forwards this message to all those who are interested in knowing about this topic. Let us take an example for further clarity. Suppose we have three machines: machine A, machine B and machine C. And say, machine A and machine B inform the MQTT Broker that they are interested to know about the topic "*today's_weather*" and machine C tells the MQTT Broker that it is interested to know about the topic "*yesterday's_weather*", now if any of the machines tell (sends a message) to the MQTT Broker anything about "today's_weather", then the MQTT Broker will simply pass this information to machine A and machine B. Thus for machine A to talk to machine B, they both need to simply be interested in a common topic, and then they can simply drop a message on the broker about this topic. In other words, if machine A wants to send "Hii" to machine B, it simply needs to tell the MQTT Broker ["Hii", "today's_weather"]. As soon as the broker gets this, it will simple send "hii" to anyone interested to know about "today's_weather", which in this case will be the machine B.

Telling the MQTT Broker that you are interested in knowing about a particular topic is called subscribing to that topic. And sending a message to the MQTT broker about that topic is called publishing a message on this topic.

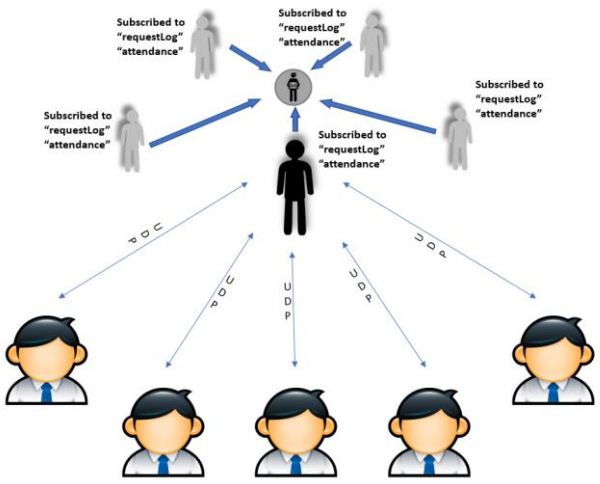

Fig 2. Architecture of the system

Now we come back to the initial system, with clients, Leader server and backup servers and the MQTT Broker. What is the role of MQTT in this system? The Servers (both Leader and Backup) communicate via the MQTT protocol. This means that in this system each Server is actually also an MQTT Client which will talk to each other using MQTT Broker and MQTT protocol.

All the servers including the leader server subscribe to the topic "currentRequestLog". This is so that the Leader server can forward the request that it received from the client to all the backup servers. The Leader server does this by simply publishing the request as a message on the "currentRequestLog", on the MQTT Broker. The broker forwards this to all its MQTT Clients, also called the Backup servers, after which all the backup servers perform this request, but of-course only the leader server sends a response to the client. This way each and every backup server gets up to date to the leader server and is always ready for taking over in case the leader server fails.

All of these servers also subscribe to the topic "*attendance*". The Leader server sends a message "*attendanceRequest*" on the "*attendance*" topic. To this each backup server has to respond with an "<IP Address>" as a published message on the "*attendance*" topic. With these IP Addresses, the leader server makes a list of all those that are there in line to be the leader in case the current leader fails. This list is forwarded as a part of the response to each client's request, so that if the leader server fails, they (other servers) will know where to go next.

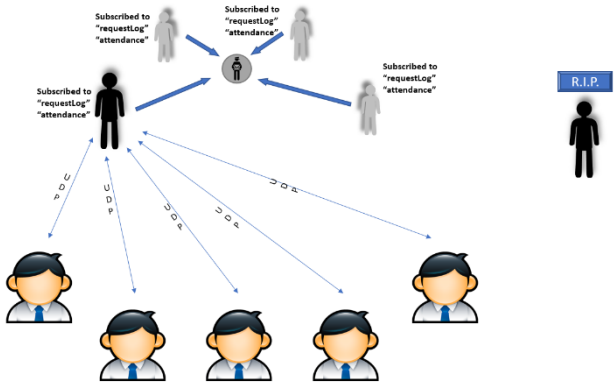

Fig 3. Failure of the Leader Server

In case the Client reaches any backup leader with its request, the first thing that the Backup server does is to ping the leader server on the topic *"<LEADER IP Address>"* where the topic is the ip address of leader server, to which only the leader server is subscribed to. Now if the leader is alive then it responds to this message and the backup server does not caters to the request of the client. But if the leader server has actually gone down and no response is received then the backup server assumes leadership and caters to the client.

IV. WORK FLOW

In this section, we dig a little deeper into the actual working of the system. We will know which messages are sent when, why; what the responses are.
First, one server starts. It subscribes to topic "LeaderTopic". Then it publishes "LeaderExists?" message to the

"LeaderTopic". It then waits for a reply from the leader(if there is one) on the same topic. As this is the first server, so no one would reply and so this would assume leadership. Then the second server goes up. This time when "LeaderExists?" message is published on the "LeaderTopic", the leader will respond to it by publishing "<Leader IP Address>" as a message to the "LeaderTopic". So this server will become a backup server. Similarly any and all servers that now start will become backup servers; thus laying the foundation of the fault tolerant system.

Now after starting the main system on the main thread, another thread runs the attendance function. In this function, the leader server publishes the message "Attendance?" on the "attendanceRequest" topic. To this message all the backup servers publish "<Their Backup IP Address>" as a message to the attendance topic. With the help of these IP Addresses, the leader server makes a list of backup servers, this is the nextLeaderList. This process is repeated every 10 seconds. Thus if there is any change in the configuration of the servers, for example, if a backup server goes down, the leader server will know about it and will have the new list in the next 10 seconds. Now whenever any client requests something from the server, the server sends the client this nextLeaderList along with the response. This is how the client knows where to go next, if the leader server fails.

Initially the client will send a UDP request message to the leader server. When this happens, the leader server, other than catering to this request, publishes this request as a message to the "currentRequestLog" topic. This request is, then, also performed by all the backup servers. Hence the state of all the servers are eventually consistent.

If the leader server drops any client's request 5 times, then the client assumes that the leader has gone down. So it pings the next server in line to be the leader. When this happens, the backup server pings the leader server by publishing a "There?" message on the topic "<Leader IP>". So to this, if the leader is still alive then it responds with a "Yes" message. If the previous leader is alive, then it responds to the client with the nextLeaderList. If, on the other hand it has really gone down then the backup server assumes leadership.

V. Challenges Faced

The main problem with programming a software with more than one point of access is understanding the work flow. This is because the same program has to run for both: in this case - the leader and the backup server; So in the actual designing it is hard to imagine which messages go where. For example, in a normal client server program, we have a message which is sent from one program to another. But in this case, for the server to server communication, we have to send a message from one part of the server code, and we know that it will be received on a different part of the same program, so we need to code that too. Understanding this flow of sending a message and receiving it on the same program in some other part is a little typical.

Setting up the testing environment is also a difficult task. This is because a network of computers has to be made and a specific role has to be assigned to each.

In this system many computers become servers. When more than one computer is involved in doing one particular job, debugging a problem becomes extremely difficult, as it is difficult to guess which system is actually causing the error.

VI. Experimental Setup

We need three or more computers: one for leader server, one for backup server and then some for clients.

There are the steps for running the server:

1. Install git.
2. Git clone https://github.com/eclipse/paho.mqtt.c.git
3. cd paho.mqtt.c
4. sudo apt-get install libssl-dev
5. make
6. sudo make install
7. cd to project directory
8. run ./trysub <SERVER IP> <PORT> <MQTT SERVER IP> <DEBUG FLAG>

For running the client, simply run the following in the terminal in the project directory:

1. ./UDPEchoClient <SERVER IP> <PORT>

VII. Conclusions

This is a fault tolerant file locking system. We use a pub-sub service for a stable connection between the servers so that although they don't know each other, they will still be able to maintain contact. With this, we come to the conclusion of the project, having built the fault tolerant file locking service successfully.

VIII. Future Work

If a server that had gone down previously, comes up then it should be able to continue as a backup server from now on. This feature is something which should be there in the project.

The only question still left unanswered is that what if the MQTT Broker itself fails? There is a simple solution for this. We can have multiple MQTT Brokers running on multiple machines, and in the case that the MQTT Broker fails, the servers can go to some other broker, thus still remaining up.

IX. Acknowledgment

I wish to thank the following people for their contributions. Prof. Violet Syrotiuk, for helping and guiding me through the project plan and for later development of the same. Diego Ongaro and John Ousterhout, for the raft Consensus Algorithm. Mr. Anubhav Gupta, who helped me understand the MQTT protocol from within and also for his invaluable support. Mr Deepak Gupta and Miss Aayushi

Gupta, for his immense help with the documentation. Lastly, the people at IBM for developing the MQTT protocol.

X. REFERENCES

[1] MQTT Organization Website, http://mqtt.org/, main article on the home page.

[2] MQTT Wiki, *"https://en.wikipedia.org/wiki/MQTT".*

[3] Raft paper, "In Search of an Understandable Consensus Algorithm".