IMPLEMENTING A CONSENSUS ALGORITHM TO ENHANCE THE STREAMING PLATFORM

Peng WANG

Mengna ZHU

Yunfeiyue LIU

(TEAM THREE)

Acknowledgements

We are deeply grateful to our project instructor and advisor, Prof. Ming-Hwa Wang, for both his teaching in the Distributed Computing class and his support to this project.

We are also very thankful to all the researchers and developers who have laid the theoretical and practical foundations for this project.

Abstract

In a distributed system like the setting of the streaming platform in our P2 or more complex, the consistency problem should be properly addressed in order to guarantee that the requests from the clients can be correctly processed by the system as long as a majority of the servers are up and can effectively communicate with each other. The distributed consensus algorithms are used to maintain the consistency of the replicated logs among the servers, here the replicated log records all the requests/commands sent from all the clients in chronological order and is stored on the local disk.

We studied the mainstream consensus algorithms including Paxos and RAFT, and reached a conclusion about both advantages and disadvantages of these consensus algorithms as well as why Raft is efficient and easier to understand. Then we implemented the Raft algorithm by simulating the problems of leader election, log replication and safety issues (e.g. leader completeness). Finally, we stretched our work by integrating the Raft algorithm as a consensus module into the streaming platform of our previous programming assignment P2 and investigate how the consensus algorithm both improves the safety and enhances the performance of the streaming platform. After deploying the consensus module, we demonstrated that the correctness and efficiency of the streaming platform has been improved in cases when a server crashes/restarts, servers are newly added to the streaming platform, communication between the server and the client is abnormal, etc.

Table of Contents

1 Introduction

1.1. Objective

There are three objectives for this study. First, it aims to get a deep understanding of consensus algorithms by comparing the Paxos algorithm and the Raft algorithms for fault-tolerant distributed systems. Then we get a conclusion of both advantages and disadvantages of the above consensus algorithms and why Raft is efficient and easier to understand. Second, it intends to implement the Raft algorithm by simulating the problems of leader election, log replication and safety issues (e.g. leader completeness). Finally, we want to stretch our work by integrating a consensus module into our previous programming assignment P2 and investigate how the consensus algorithm both improves the safety and enhances the performance of the streaming platform.

1.2. What Is the Problem

How to maintain the consistency among the servers in a distributed system like the streaming platform in P2 in scenarios such as a server crashes/restarts, servers are newly added to the streaming platform, communication between the server and the client is abnormal? If we plan to integrate a consensus module into the distributed system to solve the consistency problem, which consensus algorithm should we choose to implement the consensus module?

1.3. Why This Project is Related to This Class

Consensus is a fundamental problem in distributed system to achieve overall system reliability. A deep understanding in consensus algorithms will help us to gain deeper understanding of the distributed systems as well as great hands-on experience in working with distributed consensus at the same time.

1.4. Why Other Approach Is No Good

As we all know, the design for the Paxos protocol is a millstone for achieving consensus in asynchronous distributed systems. However, the actual requirements necessary to build a real system, including areas such as leader election, failure detection, and log management, are not present in the Paxos specification, yet add a degree of complexity that almost always significantly alters the original protocol. This is precisely where the Raft designers correctly argue that the absence of specificity leads to great difficulty in applying Paxos to real world problems. A subsequent paper by Lamport in 2001 does an excellent job of making the original protocol accessible to practitioners, and to some degree, proposes techniques for designing a replicated log, but it stops short of being prescriptive in the way that Raft does [2].

Previously our approach to deal with the consistency problem in P2 is:

1) recover the data and information of a restarted server from the files stored on the local disk before the server's crash, or let the backup server of this restarted server to send its own data and information to this server and then copy the data and information to the restarted server.

2) let an "old" server to send the data and information to a newly added server and copy the data and information to the newly added server.

The correctness of this approach cannot be guaranteed because the completeness of the data and information of either a backup server or an "old" server mentioned above cannot be proved.

In addition, transmitting data and information among the servers will consume bandwidth especially when a significantly large volume of data has been accumulated on the streaming platform after a large number of commands/requests have been sent to the streaming platform from clients.

1.5. Why Our Approach Is Better

In this project, we choose the Raft algorithm over the Paxos Algorithm. The Paxos is too complicated for implementation and it is more about theory. Raft is a more understandable algorithm and also servers as a practical tool to build a consensus distributed system.

We will utilize a replicated log to record all the requests/commands from all the clients sent to the streaming platform and maintaining the consistency of this replicated log among all the servers via the consensus algorithm. As long as the replicated log is complete and accurate, any server can safely apply the newly added commands in its replicated log to update its own state and the correctness of its state is guaranteed.

By applying the consensus algorithm to P2, the servers only need to exchange the information of the replicated logs rather than the state information to achieve the consistency of their states, obviously the traffic (the volume of data transimitted among servers) of the new approach would be much less than our previous approach thus the latency of communication among servers would be shorten and higher efficiency of the system would be achieved.

1.6. Statement of the Problem

Our project aims to implement the Raft algorithm to simulate how the server cluster is able to achieve ‘consistency’ feature when a majority of the servers are up and running, which serves as the basis of the consensus module that we will add to the streaming platform in P2. After deploying the consensus module, the safety and efficiency of the streaming platform will be improved in cases when a server crashes/restarts, servers are newly added to the streaming platform, communication between the server and the client is abnormal, etc.

1.7. Area of Investigation

We will implement the Raft algorithm and stimulate the algorithm to investigate into following mechanisms.

1) Leader Election: how does the algorithm work to select one server as leader and choose a new leader when crash detected;
2) Log Replication: how does leader accept command from other clients and append the command to its log and replicate its log to other servers in consideration of overwriting inconsistencies;
3) Safety: how well does the servers keep log consistent.

Moreover, we would like to integrate the above algorithm into the streaming planform in our P2 and investigate how the consensus algorithm both improves the safety and enhances the performance of the streaming platform.

2. Theoretical Bases and Literature Review

2.1 State Machine

In computer science, state machine replication or state machine approach is a general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas.

A State Machine will be defined as the following tuple of values:

A set of States

A set of Inputs

A set of Outputs

A transition function (Input × State → State)

An output function (Input × State → Output)

A distinguished State called Start.

A State Machine begins at the State labeled Start. Each Input received is passed through the transition and output function to produce a new State and an Output. The State is held stable until a new Input is received, while the Output is communicated to the appropriate receiver.

A State Machine should be deterministic: multiple copies of the same State Machine begin in the Start state, and receiving the same Inputs in the same order will arrive at the same State having generated the same Outputs.

By applying the concept of State Machine to P2, we can safely assert that each and every server on the streaming platform is a state machine: the clients commands are the input of the state machine, the information of servers/clients (name or id, IP address, port number), the topics (topic name, number of partitions of each topic), records/now/offset of partitions of each topic, as well as the mappings between partitions and servers/subscribers (which partition is located on which server, which client is the subscriber of a specific partition) are the states of the machines (which is deterministic by the sequence of commands). Given an input command combined with the current state, a server will update its state with the transitional function and generate the output with the output function, then reply the client with the output.

2.2 Consensus Algorithm

The consensus algorithm is used to achieve consistency among the replicated state machines on a given value in fault-tolerant distributed systems.

The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

The consensus algorithm is widely applied in real-world scenarios of distributed systems. For example, the Google infrastructure is constructed as a set of distributed services offering core functionality to developers. Chubby is an important module in the Google infrastructure, which is a fault-tolerant system that provides a distributed locking mechanism and stores small files. Here Paxos, a famous consensus algorithm, is the heart of Chubby.

2.2.1 Paxos Algorithm

Paxos is a distributed consensus algorithm, which basically consists of three steps.

Step 1, Elect a coordinator, or master from the replicas. the candidate replica will send a prepare request to the replicas. The algorithm relies on an ability to elect a coordinator for a given consensus decision. Recognizing that coordinators can fail, a flexible election process is adopted that can result in multiple coordinators coexisting, old and new, with the goal of recognizing and rejecting messages from old coordinators. To identify the right coordinator, an ordering is given to coordinators through the attachment of a sequence number. Each replica maintains the highest sequence number seen so far and, if bidding to be a coordinator, will pick a higher unique number and broadcast this to all replicas in a propose message.

If other replicas have not seen a higher bidder, they either reply with a promise message indicating that they promise to not deal with other (that is, older) coordinators with lower sequence numbers, or they send a negative acknowledgement indicating they will not vote for this coordinator. Each promise message also contains the most recent value the sender has heard as a proposal for consensus; this value may be null if no other proposals have been observed. If a majority of replica replied to the request with a promise message, the receiving replica is elected as a coordinator, with the majority of replicas supporting this coordinator known as the quorum.

Step 2, The elected coordinator must select a value and subsequently send an accept message with this value to the associated quorum. If any of the promise messages contained a value, then the coordinator must pick a value (any value) from the set of values it has received; otherwise, the coordinator is free to select its own value. Any member of the quorum that receives the accept message must accept the value and then acknowledge the acceptance. The coordinator waits, possibly indefinitely in the algorithm, for a majority of replicas to acknowledge the accept message.

Step 3: If a majority of replicas do acknowledge, then a consensus has effectively been achieved. The coordinator then broadcasts a commit message to notify replicas of this agreement. If not, then the coordinator abandons the proposal and starts again. If not, then the coordinator abandons the proposal and starts again.

The Paxos looks pretty simple. However, as mentioned in the Chubby paper, "While Paxos can be described with a page of pseudo-code, our complete implementation contains several thousand lines of C++ code."

In addition, in the RAFT paper, the author pointed out that Paxos has two significant drawbacks:

1) Paxos is exceptionally difficult to understand.

2) The second problem with Paxos is that it does not provide a good foundation for building practical implementations.

2.2.2 Raft Algorithm

Raft is similar to Paxos and it is easier to understand than Paxos. It provides the same result as (multi-) Paxos and also as efficient as Paxos. It simplifies Paxos by problem decomposition to separate the entire problem into independent parts. It also minimizes the state space by handling multiple problems with a single mechanism, eliminating the special cases, maximizing coherence and minimizing nondeterminism.

Step 1: Leader election. There are three roles in raft algorithm, which are leader, candidate and followers. Each server has a random timeout if there is no communication with other servers, when the timeout expires; it becomes a candidate and immediately increases its current term and vote for itself as the next leader. It will send RequestVote RPCs to all the other servers. There are three possible situations. First, candidate server gets votes from majority servers and become the next leader. After it becomes the next leader, it sends heartbeats to all servers in the cluster. The second one is one of the servers becomes the next leader which force the candidate to get back to follower. The last possibility is no leader is selected. If multiple servers split the votes which makes nobody gets the majority votes and no leader will be selected for this case. Since this process uses randomized timeout to determine the leader, it is much simpler than ranking in Paxos.

Step 2: Normal Operation. After leader is selected, each time client sends commands to one of the servers in the cluster. If server is a follower, it will directly forward the request to the current leader. After server receives client request, it appends command to its log and then send AppendEntries RPCs to all followers. If the new entry is committed, the leader executes command and return result to client. The leader will also notify followers and followers will execute committed commands to keep the log consistency. If the followers crashed during the process, the leader will continue trying until succeed. Since all the followers update their log entries based on the copies of their current leader, this one way replicate will also simplify the algorithm comparing to Paxos.

In order to keep the accuracy of log entries in leader server, when candidate sending out vote requests, it will also send out its last log entries to all voting machines. By comparing the length of the log entries in voting machine with the entries from candidate, the voting machine will send decline if it contains more complete log entries than candidates and send accept otherwise.

2.3 Consistency Problem in P2

Previously when I was working on P2, I didn't yet know the concept of consensus algorithm as well as the replicated state machines, so my approach to maintain the consistency of the states among all the servers was very clumsy:

1) when a server already on the streaming platform crashed and then restarted. It's possible that during the interval between its crash and restart some commands from the clients had already been sent to the streaming platform. If we simply recover the state of the restarted server using the state information stored on the disk before its crash, obviously, that "old" state information doesn't reflect the change of the state, which is produced by applying the commands sent to the streaming platform after the server's crash and before its restart to the "old" state, and thus the state information is incomplete. A different approach is to let the state information be recovered

by another server, which should be the backup server of this restarted server. However, we cannot guarantee that the state information of the backup server is absolutely complete because the backup server may be a server added to the streaming platform recently, so it cannot be guaranteed that its state information will reflect all the commands previously sent to the streaming platform as this backup server obviously doesn't know any commands sent to the streaming platform before it was added to streaming platform.

2) when new servers are added to the streaming platform, previously my algorithm handles this situation as below:

first let all the servers update their [servers] to contain the information of all the servers currently on the streaming platform. ([servers] is a Java ArrayList which stores the information of all the servers on the streaming platform, the information includes the name/id, IP address, port number of a server)

In addition, the clients (except the one who originated the [add] command) currently on streaming platform will also be notified with the information of the newly added servers. This is achieved by let the first server in [servers] (who is assumed to be on and has already received the [add] command) forward the [add] command to the clients via the [listen] channel (the connection between a server and a client which was setup when a client joined the streaming platform).

As a server newly added to the streaming platform doesn't receive any of the commands previous sent to the streaming platform before this server was added to the platform, it's still on the start state, or in other words, it doesn't have any knowledge about the topics, partitions, mappings. However, we need to update this server's state to the current one. To achieve this, my previous approach was letting an "old" server to send the state information to this newly added server and the newly added server will use it to update its state (this operation happens when an old server relocates one of its partition to the newly added server). However, we cannot guarantee the "old" server's state information is complete because that "old" server may not be the "oldest" server on the streaming platform and therefore it's possible that this "old" server also missed some commands sent from the clients. Furthermore, even this "old" server is one of the oldest server, this server possibly experienced crash and restart so it could also miss the commands sent from the clients during the period between its crash and restart and therefore its state information might possibly also incomplete.

2.4 Employ the consensus algorithm to solve the consistency problem in P2

I was very confused by the situations analyzed above and have to admit I cannot guarantee the correctness of my approach to deal with the consistency problem in P2. After I learnt the concept of consensus algorithm and replicated state machines, however, I believe the problem of P2 could be better attacked by utilizing a replicated log to record all the commands from all the clients sent to the streaming platform (we will also update and record the state information after applying a command to the state machine, both the replicated log and the current state should also be stored on the disk to survive the possible crash of the server) and maintaining the consistency of this replicated log among all the servers via the consensus algorithm. As long as the replicated log is complete and accurate, any server can safely apply the newly added commands in its replicated log to update its own state and the correctness of its state is guaranteed.

By employing the consensus algorithm to P2, servers only need to exchange information of the replicated logs rather than the state information to achieve the consistency of their states, obviously the traffic (the volume of data transmitted among servers) of the former would be much less than the latter. Thus the latency of communication among servers would be shortened and higher efficiency of the system would be achieved.

3. Hypothesis

1. The server on the streaming platform of P2 can be implemented as the state machine, the commands sent from the clients are replicated to all the servers in its replicated log, each command is a log entry.

2. A consensus module can be added to the streaming platform, which is essentially a consensus algorithm, to maintain the consistency of the replicated logs among all the servers on the streaming platform.

3. As long as the consistency of the replicated logs among all the servers is maintained by the consensus module, the correctness of the state of each server as well as the output it generates in response to the request/command of the client can be guaranteed no matter if a particular server is newly added to the streaming platform or it just experienced crash and restart.

4. Compared with the previous approach of copy the state to the restarted/newly added server from an “old” server, we only need to transmit the entries (commands) of the replicated log between servers if the consensus module is deployed to the streaming platform. Therefore, the volume of data transmitted among servers would be significantly less in the latter case than that in the former case, so the bandwidth will be saved, the latency of communication would be shorten and higher efficiency of the system would be achieved.

4. Methodology

We will first try to implement a consensus algorithm and then try to apply this consensus algorithm to P2 for its improvement.

4.1. Generate Input and Output

The servers in our P2 are state machines: the clients commands are the input of the state machine, the information of servers/clients (name or id, IP address, port number), the topics (topic name, number of partitions of each topic), records/now/offset of partitions of each topic, as well as the mappings between partitions and servers/subscribers (which partition is located on which server, which client is the subscriber of a specific partition) are the **states** of the machines (which is deterministic by the sequence of commands). Given an input command combined with the current state, a server will update its state with the transitional function and generate the **output** with the output function, then reply the client with the output.

With such a perspective, the **log of all the commands** (with correct chronological order) sent by all the clients can be replicated on all the servers and stored on the local disk together with the current state of the server, no matter which server is crashed/restarted, which server is newly added to/deleted from the streaming platform, we can always utilize the consensus algorithm to **guarantee the consistency of the replicated log**, and have only a leader, or a master server (the sever elected via the consensus algorithm) to handle the client command, generate the output and send it back to the client in most cases. If the master server fails, a new master is automatically elected, which will then continue to serve clients based on the contents of its local copy of the replicated log and its current state.

The command from a client, together with the client name (id), and a serial number would be recorded in the replicated log as an entry. For example, a [create] command from a client [c2] with a serial number 3 (the 3rd command originated from [c2]) would be recorded as [c2 3 create (topic=t2 partitions=3)]. Here the serial number is used to avoid the repeated processing of the same command by servers in the case that a master server (the leader) crashes when it is processing a command and then a newly selected master server processes the same command twice.

4.2. The Consensus Algorithm

We will employ a consensus algorithm to maintain the consistency of the replicated log. The brief description of the consensus algorithm is as below:

1. Elect a master server or the leader. Upon being elected as the leader, the master server will immediately send the heartbeat RPC to all other servers as the notification of existence of the master server.
2. The master server will be responsible for receiving the command from the clients. If a command is sent to a server (the follower) other than the master server. The follower will redirect the command to the leader. The leader will first append the received command to its own log, then replicate the command to all other servers' replicated log via the [appendEntry RPC]. If the command has already been replicated to a majority of the servers, this command is considered as "committed". After a command has been committed, the master server will apply the command to its own state machine and send the output to the client as the reply. However, here in P2 the situation may be a little tricky because the state of each state machine may not be exactly the same because the partitions of topics are distributed on the servers rather than equally copied to all the servers. For example, a server [s1] may have only the [partition 0] of topic [t1], and a server [s2] may only have the [partition 1] of topic [t1]. However, the information of partitions and topics is a part of the state. In such a case, the state of [s1] is different from that of [s2]. If a client send a [get] command like [get (topic=t1 partition=0)], assuming currently [s2] is the leader, if the command is processed by [s2], it will generate an output of [partition 0 of t1 is NOT located on s2] and send it back to client. Only [s1] can get the record on [partition 0] of topic [t1] (for example, a record like [abc 3]) and send this record back to the client. For such a reason, the master server may need to find which server should be responsible for a specific command and then require that server to process this command and reply the client. Anyway, any server can only reply the client when its replicated log is up-to-date, or in other words, exactly the same as that of the master server.
3. In the case the master is crashed, a follow server will become a candidate of the master server and send the [requestForVote RPC] to all other servers. If this candidate receives

votes from a majority of the servers, it would be elected as the new leader. The consensus algorithm will guarantee that only the candidate with the most up-to-date log will be elected as the new leader.

We plan to deploy 5 servers on the streaming platform and according to the safety property of the consensus algorithm, as long as 3 out of 5 servers is on, the streaming platform will always work properly to handle the commands from the clients.

4.3. Language and Tools Used

We will implement the consensus algorithm with Java.

We will use at least three machines as the hosts of the servers, each machine has a unique IP address. For example, we can use our own laptop to host the server program besides the two ITU machines, in such a case, we may possibly have 5 machines if all three team members can use his/her own machine with different IP addresses at different locations, and each machine will host one server program.

4.4. How to Test Against Hypothesis

We will prepare a sequence of commands in a text file (command file) for the client to send to the streaming platform as input of the system.

The state of the server will be recorded and stored as data and info files on the local disk.

The commands/requests of clients will be recorded in the log and stored on the local disk.

We will initialize three servers in the cluster at first, run client commands to create/subscribe topics and publish records. Then we will:

1) simulate the crash and restart a server by employing the [Random] class of Java. Before the server gets back, send client commands to the streaming platform. Then restart the server.
2) add two more servers to the streaming platform and continue sending client commands to the streaming platform.

We will test the hypothesis by:

1) Comparing the replicated log of the master server (the leader) with the previously prepared command file.
2) Comparing the replicated logs of the restarted/newly added servers with that of the master server.
3) Checking the output by displaying it in the client window.
4) Checking the data and info files of the servers.
5) Recording and comparing the traffic among servers before and after deploying the consensus module.

With above tests we will check the correctness of the implemented consensus algorithm and investigate if the consensus module can help to improve the safety and efficiency of the streaming platform.

5. Implementation

We implemented Raft algorithm and integrated it into the streaming platform as the consensus module to solve the consistency problem in the programming assignment P2. The programming language we used is Java.

5.1. Code

The code is available from https://github.com/gladet/stream_raft_v7

5.2. Programming Design and Flowchart

2.4.1 Programming Design

In the implementation of Raft algorithm. We developed the below Java classes:

LogEntry: defines the data structure of a log entry as well as the operations on the entry

RaftData: defines a set of data used in the Raft algorithm as well as the operations on these data

FollowerThread: defines the behavior of a follower

CandidateThread: defines the behavior of a candidate

LeaderThread: defines the behavior of the leader

In addition, we add new methods to the already existed class ServerThread of the streaming platform, which is the core module of the streaming platform for the server-server and server-client communication:

procReqVote: processes the VoteRequest RPC

procReplyVote: processes the reply to the VoteRequest RPC

procAppend: processes the Append RPC

procReplyAppend: processes the reply to the Append RPC

We also modified the server, ServerThread, client and ClientThread class of the streaming platform to adapt them to the newly added consensus module.

2.4.2 Flowchart Description

The flowchart is in the Appendix A.

System Initialization

1.1 Initially we start the server programs on different machines.

1.2 Then we start a client program and send the add command to the server programs. As add command is regarded as a system configuration command, it will be executed immediately by all the servers receiving this command and it will not be appended to the log.

Leader Election

2.1 As long as a server has been added to the streaming platform, it will immediately step into the FOLLOWER state waiting for the Heartbeat or Append RPC from the leader.

2.2 If the follower doesn't receive any RPC message from the leader within a predefined TIMEOUT interval, it will switch to the CANDIDATE state and initiate an election by sending VoteRequest RPCs to other servers on the streaming platform.

2.3 When a server receives the VoteRequest RPC from a candidate, it will grant the vote to the candidate only when this server doesn't yet vote for another candidate and the candidate is at least as up-to-date as this server.

2.4 If the candidate receives votes from a majority of servers within a predefined TIMEOUT interval, it will switch to the LEADER state. If the candidate receives a Heartbeat or Append RPC from the leader, it will switch back to the FOLLOWER state. If neither of these two situations happens within the predefined period of time, the candidate will reset the CANDIDATE cycle and resend VoteRequest RPCs to other servers.

2.5 If a candidate has successfully won the election and become the leader, it will notify all other servers by sending a Heartbeat RPC, which essentially is an Append RPC but without any log entries in the message.

Log Replication

To test the effectiveness of Raft algorithm as the consensus module added to the streaming platform, we send client requests like create/publish/subscribe via the client program to all the servers on the streaming platform.

3.1 If a follower receives such a request, it will just ignore it without doing anything except printing out the message. If the leader receives such a request, it will immediately append the message as a new entry to its log and write the updated log to the log file on the local disk. After that the leader will initiate an Append RPC to send the newly added log entry to the followers.

3.2 When a follower receives such an Append RPC from the leader, it will replicate the log entries contained in the Append RPC to its own log if the log entry preceding the first to-be-replicated log entry in the leader's log matches the log entry with the same log index in this follower's log, and send an ACK message to notify the leader about the success of the Append RPC. Otherwise the follower will send a negative ACK message back to the leader to notify the leader about the failure of the Append RPC.

3.3 If the leader receives a negative ACK message from a follower, it will decrement the index of the first to-be-replicated log entry and send a new Append RPC to that follower. Such a dialog will be repeated until the specified log entries in both parties' logs matches and the follower finally successfully replicate the log entries to its log and send an ACK message to the leader.

3.4 When a leader receives an ACK message from a follower, it will also check if there is a log entry has been successfully replicated to a majority of the servers, such a log entry is defined as being committed. As long as a log entry is newly committed, the leader will execute the client request contained in this log entry and reply the client with the output of the execution of the request. In addition, the leader will broadcast the index of the newly committed log entry to all the followers via the Append RPC.

3.5 When a follower hears about this newly committed log entry, it will also execute the client request contained in this log entry and reply the client with the output of the execution.

6. Data Analysis and Discussion

6.1. Test for Output Generation and Output Analysis

In the Raft paper, the parameter setting is:

Follower timeout: randomly generated, in the interval of 150 – 300 milliseconds

Candidate timeout: randomly generated, in the interval of 150 – 300 milliseconds

Leader Append RPC frequency: 1 RPC per 40 milliseconds

For the purpose of test and demonstration, we set the parameters as below:

Follower timeout: randomly generated, in the interval of 7 – 14 seconds

Candidate timeout: randomly generated, in the interval of 7 – 14 seconds

Leader Append RPC frequency: 1 RPC per 3 seconds

Initially we start the server programs on three machines. The IP addresses of the machines are: 68.181.201.26, 23.253.20.67, 104.130.67.11.

Then we start a client program and input the add command to add one server to the streaming platform. As long as this server has been added to the streaming platform, it will immediately step into the FOLLOWER state waiting for the Heartbeat or Append RPC from the leader.

As this server is currently the only server on the streaming platform, it will experience FOLLOWER TIMEOUT and switch to the CANDIDATE state, then it will initiate an election and of course it wins the election after voting for itself and then becomes the leader.

Now the client send requests to the streaming platform. The leader wrap each request into a log entry and appends the log entry to its log, then the log entries are committed and the client requests enclosed in the log entries are executed by this leader as it’s the only server on the streaming platform. The outputs of the execution are sent to the client, which are displayed in the client window.

Now we add two other servers to the streaming platform. These two servers will also step into the FOLLOWER state and then receive the Append RPC from the current leader, which is the first server added to the streaming platform. The two followers will replicate the log entries included in the Append RPC to their own logs and send an ACK message to the leader. We can see that currently the three servers' logs are identical. In addition, as these log entries have already been committed, the two servers will also execute the client requests enclosed in these log entries and reply the client with the outputs, which are also displayed in the client window.

We further send some commands via the client program to the streaming platform. When the leader receives such a command, it will immediately append the message as a new entry to its log and then initiate an Append RPC to send the newly added log entry to the two followers.

When a follower receives such an Append RPC from the leader, it will replicate the log entry to its own log and send an ACK message to the leader. Again, we can see that currently the three servers' logs are identical.

As long as the newly replicated log entry has been committed, the command wrapped in this log entry will be executed by the three servers and the output of the execution will be sent to the client, which is displayed in the client window.

Now we simulate the crash of the leader by terminating it. After that one of the follower will experience the TIMEOUT earlier than the other one and switches to the CANDIDATE state. This candidate initiates an election by sending VoteRequest RPCs to servers on the streaming platform, which include the other server and itself. This candidate wins the election after receives votes from both itself and the other server and becomes the new leader.

We further send some commands via the client program to the streaming platform. When the leader receives such a command, it will immediately append the message as a new entry to its log and then initiate an Append RPC to send the newly added log entry to another server on the streaming platform, who is currently the only follower.

When this follower receives such an Append RPC from the leader, it will replicate the log entry to its own log and send an ACK message to the leader. Again, we can see that currently the two servers' logs are identical. As long as the newly replicated log entry has been committed, the command wrapped in this log entry will be executed by the two servers and the output of the execution will be sent to the client, which is displayed in the client window.

Now we restart the crashed server. It comes back to the streaming platform as a follower. It will first recover its log from the log file stored on the local disk before its crash, it will also recover its state (the topics, records/now/offset of partitions of each topic, as well as the mappings between partitions and servers/subscribers) from the data and info file stored on the local disk before its crash. Then it will receive the Append RPC from the current leader and replicate the log entries included in the Append RPC to its own logs and send an ACK message to the leader, now its log is also up-to-date as those of the other servers on the streaming platform. We can see that this server's log is identical to those of the other two servers now. In addition, as these log entries have already been committed, this server will also execute the client requests enclosed in these log entries and reply the client with the outputs, which are displayed in the client window.

6.2. Compare Output Against Hypothesis

1. The server on the streaming platform of P2 can be implemented as the state machine, the commands sent from the clients are replicated to all the servers in its replicated log, each command is a log entry.

True. The commands sent from the client programs are wrapped into the log entries, the Raft algorithm we implemented can guarantee that all the log entries will eventually be replicated to all the servers' replicated logs in the same order (sequential consistency).

2. A consensus module can be added to the streaming platform, which is essentially a consensus algorithm, to maintain the consistency of the replicated logs among all the servers on the streaming platform.

True. We have implemented the Raft algorithm and integrated it into the streaming platform as the consensus module, which maintain the sequential consistency of the replicated logs among all the servers on the streaming platform.

3. As long as the consistency of the replicated logs among all the servers is maintained by the consensus module, the correctness of the state of each server as well as the output it generates in response to the request/command of the client can be guaranteed no matter if a particular server is newly added to the streaming platform or it just experienced crash and restart.

True. It's obvious that the correctness of the state of each server as well as the output it generates in response to the request/command of the client can be guaranteed when all the servers on the streaming platforms are up and running properly. We have demonstrated that after deploying the consensus module, even in the case that a particular server is newly added to the streaming platform or it just experienced crash and restart, the sequential consistency of the replicated logs among all the servers can still be maintained. Furthermore, the client requests wrapped in the log entries will be executed by each and every server after the log entries have been committed. Therefore the correctness of the state of each server as well as the output it generates in response to the request/command of the client can also be guaranteed.

4. Compared with the previous approach of copy the state to the restarted/newly added server from an “old” server, we only need to transmit the entries (commands) of the replicated log between servers if the consensus module is deployed to the streaming platform.

True. In the previous approach, when the crashed server has restarted, the topics (topic name, number of partitions of each topic), records/now/offset of partitions of each topic, as well as the mappings between partitions and servers/subscribers (which partition is located on which server, which client is the subscriber of a specific partition) will be recovered by letting the backup server of the restarted server send these data and information to the restarted server. After deploying the consensus module, the restarted server will first recover the replicated log, topics, records/now/offset of partitions of each topic, as well as the mappings between partitions and servers/subscribers from the data and info files on its local disk. Then its replicated log will be updated by the Append RPC from the current leader. Furthermore, the client commands sent to the streaming platform after this server’s crash and before its restart, which correspond to the newly added and committed log entries in its replicated log, will then be executed by this server, which guarantees that this servers state will be up-to-date.

In the case that some servers are newly added to the streaming platform, my previous approach was letting an “old” server to send the topics, records/now/offset of partitions of each topic, as well as the mappings between partitions and servers/subscribers to this newly added server and the newly added server will use these data and information to update its state (this operation happens when an old server relocates one of its partition to the newly added server). After deploying the consensus module, all the log entries in the leader’s log will be replicated to this newly added server by the Append RPC from the current leader. Furthermore, all the historical client commands sent to the streaming platform, which are wrapped in the log entries, will then be executed by this server in exactly the same order as that being executed by any other server on the streaming platform, which guarantees that this servers state will also be up-to-date.

6.3. Abnormal Case Explanation

Currently we don’t yet discover any abnormal case in our test.

6.4. Discussion

As introduced in the textbook [10] of the Distributed Computing class, there are three ways for update propagation among the replicas:

1. Propagate only a notification of an update
2. Transfer data from one copy to another
3. Propagate the update operation to other copies

Our previous approach is the second one above for handling the cases when a server crashes/restarts, servers are newly added to the streaming platform. However, after deploying the consensus module, we employ the third approach, which is not to transfer any data modifications at all, but to tell each replica which update operation it should perform. This approach, also referred to as active replication, assumes that each replica is represented by a process capable of “actively” keeping its associated data up to date by performing operations (Schneider, 1990). The main benefit of active replication is that updates can often be propagated at minimal bandwidth costs, provided the size of the parameters associated with an operation are relatively small, which exactly matches our case of log replication. On the other hand, more processing power may be required by each replica, especially when operations are relatively complex.

Due to the time limitation, we don’t implement the comparison of the traffic of communication among all the servers before and after the deployment of the consensus module. An intuitive way for implementation is assign a global variable to record the traffic and simply add any new traffic to it when a server sends a message to another server. Here the traffic can be measured by total number of characters in each message as the message is essentially a line of multiple lines of strings.

7. Conclusions and Recommendations

7.1. Summary and Conclusions

In a distributed system like the setting of the streaming platform in our P2 or more complex, the consistency problem should be properly addressed in order to guarantee that the requests from the clients can be correctly processed by the system as long as a majority of the servers are up and can effectively communicate with each other. The distributed consensus algorithms are used to maintain the consistency of the replicated logs among the servers, here the replicated log records all the requests/commands sent from all the clients in chronological order and is stored on the local disk.

We studied the mainstream consensus algorithms including Paxos and RAFT, and reached a conclusion about both advantages and disadvantages of these consensus algorithms as well as why Raft is efficient and easier to understand. Then we implemented the Raft algorithm by simulating the problems of leader election, log replication and safety issues (e.g. leader completeness). Finally, we stretched our work by integrating the Raft algorithm as a consensus module into the streaming platform of our previous programming assignment P2 and investigate how the consensus algorithm both improves the safety and enhances the performance of the streaming platform.

After deploying the consensus module, we demonstrated that the correctness and efficiency of the streaming platform has been improved in cases when a server crashes/restarts, servers are newly added to the streaming platform, communication between the server and the client is abnormal, etc. We also tested the enhanced version of the streaming platform with integration of the consensus module against the previously set hypotheses and proved the correctness of these hypotheses.

7.2. Recommendations for Future Studies

We really enjoyed this team project because it provided us a great opportunity to learn and practice how to follow the standard research procedure to identify significant research or application problems and leverage the academic and research resources in the effort of devising innovative and effective solutions to solve those problems. Specifically, in this team project we implemented a consensus algorithm called Raft, which was invented by the Ph.D. student (at that time) and professor of Stanford University and has gained significant popularity recently, and integrated it to the streaming platform of our programing assignments in the Distributed Computing class as the consensus module to solve the previously existed consistency problem. We gained a feeling of achievement when we demonstrated that the correctness and efficiency of the streaming platform has been improved with the integrated consensus module just as we previously envisioned before our implementation.

In addition, in this project we gained deeper understanding of several important topics taught in the Distributed Computing class including Election Algorithms (in Chapter 5: Synchronization), Consistency Protocols (in Chapter 6: Consistency and Replication) and Distributed Commit (in Chapter 7: Fault Tolerance) after we implemented algorithms corresponding to these topics. Such a valuable experience further enhanced our belief that the best way of learning something is by doing.

As pointed out in the original Raft paper, Raft's log grows during normal operation to incorporate more client requests, but in a practical system, it cannot grow without bound. As the log grows longer, it occupies more space and takes more time to replay. This will eventually cause availability problems without some mechanism to discard obsolete information that has accumulated in the log. Due to time limitation, we don't yet implement log compaction in the team project. We will further study the log compaction section in the paper and try to implement this function and add it to the consensus module of the streaming platform for better system performance.

We also consider to further study and then implement the Paxos algorithm with the purpose to explore if this algorithm is really "so difficult" to understand and implement as commented by many people.

8. Reference

[1] The Raft Consensus Algorithm. Retrieved from https://raft.github.io/

[2] The Raft Protocol: A Better Paxos?. Retrieved from http://engineering.cerner.com/2014/01/the-raft-protocol-a-better-paxos/

[3] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective. In *Proc. PODC'07, ACM Symposium on Principles of Distributed Computing (2007)*, ACM, pp. 398–407

[4] Coulouris, G. Designing Distributed Systems: Google Case Study. *Distributed Systems: Concepts and Design (5th Ed)*

[5] LAMPORT, L. Fast paxos. *Distributed Computing 19, 2 (2006)*, 79–103.

[6] VAN RENESSE, R. Paxos made moderately complex. *Tech. rep., Cornell University, 2012*

[7] Lamport, Leslie, and Mike Massa. Cheap paxos. Dependable Systems and Networks, 2004 *International Conference on. IEEE, 2004*

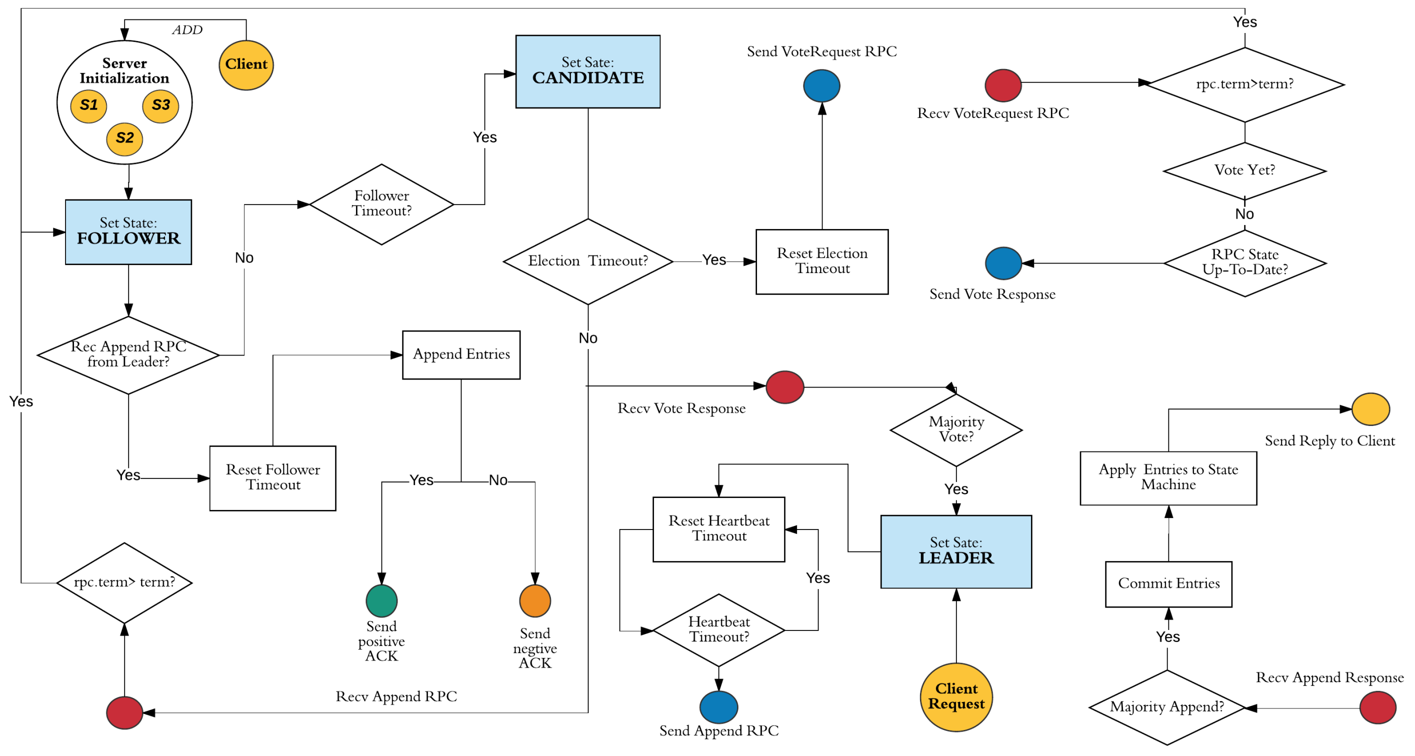
[8] Ongaro, D., and Ousterhout, J. In Search of an Understandable Consensus Algorithm. In *Proc. ATC'14, USENIX Annual Technical Conference (2014), USENIX. ii*

[9] LAMPORT, L. Paxos made simple. *ACM SIGACT News 32*, 4 (Dec. 2001), 18–25.

[10] Andrew S. Tanenbaum, Maarten van Steen. (2006). *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice Hall,

[11] Schineider, F.: "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial." *ACM Comput. Surv*., vol.22, no. 4, pp. 299-320, Dec. 1990. Cited on page 331.

9. Appendix A



Streaming Platform Enhanced with Raft Algorithm