Project Report: a fault tolerant instant group-chat based on Raft consensus algorithm

COMP90020 Distributed Algorithm Semester 1 2024

Liao Cao
caolc1@student.unimelb.edu.au
student id: 1205512

Kwun Ming Pang
kwunmingp@student.unimelb.edu.au
student id: 1443294

***Abstract*—In this report, we will demonstrate our experimentation with a consensus algorithm for a practical problem. We will discuss rationale for using a consensus algorithm, followed by discussion on what benefits and drawbacks are introduced using the consensus algorithm.**

***Keywords*—Raft, instant group chat, distributed algorithm, Raft implementation, consensus algorithm**

I. Introduction and background

Failure of a node is inevitable in a distributed system, therefore a single server architecture is vulnerable to failure. Using multiple-servers for a service can minimize the chance of such failure, since it is unlikely that they would all fail simultaneously. However a new issue occurs with this idea; with multiple-servers, we need a way for them to be consistent and coordinate efficiently.

Consensus is known as the problem of agreement [1], that for components in a distributed system, they would agree on the same value. With a consensus algorithm, we can solve the issue mentioned above and create a consistent and fault-tolerant service with multiple-servers. To demonstrate this idea, in this report, we will showcase our implementation of a fault-tolerant instant group-chat application, built using a consensus algorithm.

In our report, we will first introduce the background survey in section II, followed by comparisons of different alternatives and rationale for deciding on Raft in section III. After that, we will discuss the possible applications of Raft algorithm and potential future works in IV. Lastly, we will talk about Raft algorithm and our implementation followed by a short analysis in section V.

II. Survey of related and direct work

Consensus algorithms are to solve the consensus problem. In order to evaluate a consensus algorithm, we can use three property:

1) safety: all nodes output the same value
2) integrity: if all nodes start with a value v_i, they should decide on that value v_i
3) liveness: algorithm will eventually terminate

Paxos algorithm [2] is one of the earliest proposed consensus algorithms. It uses a two-phase procedure for a cluster to decide on a proposed value. It guarantees safety, and integrity, but not liveness. However, Paxos only proposed how to make an agreement on a single decision in [2]. To extend Paxos, several modified versions have been proposed. Fast Paxos [3] has been proposed by Lamport based on Brasileiro et al.'s idea [4]. This method decreases the message number for reaching consensus. Another version is Multicoordinated Paxos which is more resilient and guarantees liveness without increasing the latency of a proposal procedure [5].

The emphasis of above Paxos family algorithms is on the theoretical foundations, which introduce too much complexity to be implemented in practical systems [6] [7]. In contract, there are some consensus algorithms that focus more on the practical applications and has been widely adopted in the industry.

Replicated state machine is one of the methods for consensus. As discussed in section I, the consensus problem is about deciding on a single value. Replicated state machine extends this idea and instead of agreeing on a single value, the nodes should agree on the same sequence of values.

As we discussed in lecture, the motivation for replicated state machine is to create an illusion of a single fault-tolerant service to users. The use of replicated state machine provides a way to continue to provide reliable service, even when some nodes might crash in the process.

ZooKeeper's Atomic Broadcast (ZAB) [8] is a high-performance broadcast algorithm that is crash-recovery. Another consensus algorithm is Raft [6] proposed by Diego et al., which relies heavily on the replicated state machine discussed above. It provides the same guarantees as Paxos or Multicoordinated Paxos, including correct result and latency upper bound. Diego et al. states that Raft is as good as Paxos and much more understandable. This feature is attracting an increasing number of consensus applications based on Raft. As an example, Kafka decided to replace Zookeeper with Raft in 2020 [9]. This adaptation enables them to manage metadata in a more scalable and robust way, enabling support for more partitions and simplifying the deployment and configuration of Kafka.

III. Critical analysis of the algorithms reviewed

Paxos largely impacted on consensus algorithms since proposed, and influenced lots of consensus algorithms afterward

[6]. Paxos proposes three roles, proposer, acceptor, and learner, and uses a two-phase proposal procedure to ensure safety and correctness of the algorithm. A value will be proposed by proposers and be voted by acceptors, then be learnt by learners after quorums of votings. The first phase is the prepare-promise phase, which ensures integrity and safety property of consensus problems since it guarantees that if there exists a proposed value accepted by quorums of acceptors, no second proposed value will be proposed. The second phase is accept-accepted phase, where a proposer broadcasts its proposal to acceptors and waits for acceptance from quorums of acceptors. A proposed value will be decided after phase 2, unless being interrupted by another proposer. Indeed, interruption in proposal procedure can happen for infinite times, as a FLP scenario [10], which fails for Paxos to ensure liveness in a crash-recovery asynchronous model.

Though Paxos were strongly influential in research and application development of consensus algorithms in the 00s, as Ongaro et al. [6] states, Paxos algorithm is too difficult to understand. Paxos is also criticized for its difficulty in implementation in the real world, since it leaves out many specific cases that might have occurred in practical systems. For example, Chubby built for Google File System is developed based on Paxos. Chandra et al. [7] blames that there exists a huge gap between Paxos theory and the needs for real-world application. Junqueira et al. [8] point out that Paxos can not run parallel transactions unless batching them all together, which will introduce more latency and decrease throughput. From the above claims, Paxos is not suitable for our project. For one thing, Paxos is notoriously opaque for understanding and implementation in a short period. It can be foreseen that we will spend much effort on studying the Paxos paper and struggle when implementing some unmentioned scenarios. Therefore we should instead focus on the consensus algorithms that are widely used in practice.

A. *practical consensus algorithms*

ZooKeeper's Atomic Broadcast (ZAB) [8] protocol is a high-performance broadcast algorithm that serves for Yahoo! and Apache in highly-available coordination services. ZAB might be one of the most widely used consensus algorithms in the world. Its correctness and performance are verified by millions of users and applications based on it. As Junqueira et al.'s [8] prove, ZAB ensures safety, integrity, and a total ordering of the primaries (similar to proposal in Paxos). ZAB also satisfies liveness by introducing a random delay. Compared to Paxos, ZAB's strong leadership pattern simplifies lots of logic in consensus, which is more understandable, and its detailed specification covers almost all the situations to encounter in practical systems.

However, given a short project window, engineering ZAB from scratch may pose too much of a challenge. In contrast to the ZAB, Raft is known for its simplicity and ease of implementation. Raft is a consensus algorithm proposed by [6]. A key advantage of Raft is despite being much simpler for understanding and implementing, it also provides at least the same guarantee compared to Paxos. Raft will be further discussed in section V-A.

Comparing ZAB and Raft, Raft has only two types of communication, which only needs four types of message to be implemented, while ZAB has 10 message types. Though ZAB will be more efficient by using different messages in specific situations, it introduces lots of work load for coding and testing for each message type and their state machine. Using a strong leader pattern for both ZAB and Raft, Raft only cares about the data flow from leader to followers, while ZAB will take care of data flow in bi-direction in order to be more sensitive in detection of faults. For our application, Raft's guarantee in failure detection is sufficient, therefore it is unnecessary to implement a complex algorithm like ZAB. When a leader fails and a new leader comes to power, ZAB will apply a full version of synchronization of log replicas while Raft will only apply an incremental part of synchronization between new leader and followers. Raft can recover faster than ZAB if most of the followers' log replicas are up-to-date. For above claims, we finally choose Raft as the consensus algorithm for our chat application.

IV. Future directions and applications

Raft algorithm can be applied to a range of problems. Specifically, Raft is designed to solve the issues of practical distributed systems with minimal complexity. Raft utilize the method of replicated state machine [6] to provide fault tolerance, that is especially important to a large scale application. Possible application for a large scale application of Raft is to manage leader election, as the coordinator for an application [6]. Additionally, Raft can be used to manage the configuration for a distributed system so that different components can maintain consistent configurations.

A. *Case study*

RabbitMQ is a very popular open-source application, that works as message broker to exchange messages between different services. One of the message broker services provided by RabbitMQ is the Quorum Queues [11], that is designed to be highly durable and fault-tolerant, built using the Raft algorithm. Raft algorithm allows the Quorum Queues to be highly resilient to data loss [11], which may be significant to some critical sections of application e.g. the bank transaction. However, the caveat for using the Quorum Queues is that Raft algorithm may introduce unnecessary latency to the application due to the log exchange nature of the application, and the large log that the algorithm can produce over time [11]. This is not desirable in some application like real-time gaming, where low latency is very important, and the data are mostly transient.

B. *Future directions*

Since there are a wide-range of applications of Raft algorithm, our future directions includes attempting to apply it to other types of applications and looking to explore further with the Raft algorithm. In terms of applications, we would

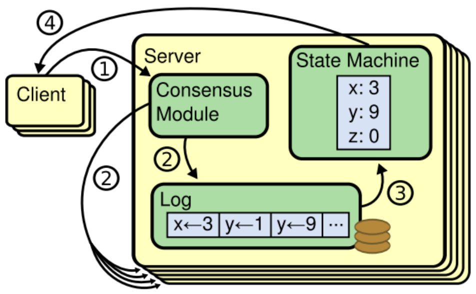


Fig. 1: From [6], visualization of state machine

like to explore the sharing of configuration in a distributed system using Raft algorithm. In terms of exploring further with Raft algorithm, there are many extensions to the basic Raft algorithm, like membership changes and log compaction mentioned in [6], which we would like to apply in our application to extend the functionality as well.

In specific, membership changes allow the services to continue providing services during the changes of configuration, like adding or removing some running instances [6]. In our application, we implemented a closed group membership, where we defined a group of servers in the configuration and the application cannot adapt to a new configuration until they are stopped. The idea of membership changes that allows the configuration to change without stopping the server is a huge improvement if we were to deploy our application.

Log compaction is a technique to solve the issue machine crashing caused by a excessively large log file [6]. Since for our application, the amount of logs is very limited compared to an actual deployed application that millions of users can use everyday, log compaction was not very necessary. Nonetheless we are interested to explore the compaction algorithm further to prepare for a large scale application.

V. Discussion of chosen algorithm, implementation details and formal description

The core idea of Raft algorithm is to utilize replicated state machine (see fig 1), which provide an easy framework to implement a fault-tolerant distributed system [12].

The construction of replicated state machine relies heavily on the deterministic property of algorithms, which states that given the same sequence of inputs, the outputs are always the same [6]. Therefore, given consistent log replication which can be viewed as inputs, different nodes in the system should produce consistent state machines.

The replicated states can be anything that suits the application. For a large-scale application, the replicated states can for example be a configuration, such that all components in the system can share the same configuration. An example will be the KRaft (Apache Kafka Raft) mentioned in section II; Kafka uses Raft algorithm to share configuration across the entire system [13].

A. Raft algorithm discussion

The role of Raft algorithm is to provide a simple framework for replicating consistent logs on different nodes in the system, so that the identical state machines can be constructed deterministically as we discussed previously. Raft uses a strong leader approach, such that the log entries flow unidirectionally from the strong leader to other nodes, called the followers, which aims to simplify the process of log replication [6]. Additionally, Raft decomposes the consensus problem into three sub-problems being:

1) Leader election: a new leader should be elected when the current leader fails.
2) Log replication: the elected leader will forward the received logs to all the nodes in the system.
3) Safety: ensures consistent states across all nodes. If any node applies a log at a particular index, all nodes should apply the same log at that index.

The simplicity of Raft algorithm lies in the fact that it requires minimally 2 remote procedure endpoints to implement the basic functions. On top of that, the total amount of states required by Raft algorithm is also as simplified as possible [6]. Based on that, we would say that Raft provides a simple framework for a very complex problem. Additionally, based our experimentation with Raft algorithm, we agree that Raft succeed in accomplishing the ultimate goal mentioned in [6], that they provide "understandability" for a large-scale audience.

The 2 remote procedure endpoints are

1) AppendEntries RPC: invoked by the leader to replicate logs to its followers, which also serves as heartbeat for failure detection. If any follower has't received an AppendEntries request from the leader for a given amount of time, they consider the leader crashed and start the election process (see (1) in fig 2).
2) RequestVote RPC: invoked by the candidate during the election process to receive votes from other nodes in the system.

There are 3 different types of roles in Raft

1) Leader: coordinator of the system, responsible for replicating logs.
2) Follower: receives log from leader and detects leader failure.
3) Candidate: candidate for potential leader; the candidate with the highest term value should be elected as leader (discussed below).

The transition of these roles can be visualized in fig 2. Transition (1) happens when the follower detects the failure of the leader, by implementing a random internal election timer for each node. The random element of the election timer is designed to "break ties"; this prevents all nodes from timing out simultaneously. In expectation, at least some nodes will have a shorter timeout interval than other nodes, and win the vote from the other nodes, preventing the election process from running indefinitely. In the event of such an unfortunate "tie" situation, transition (3) will occur that is also implemented

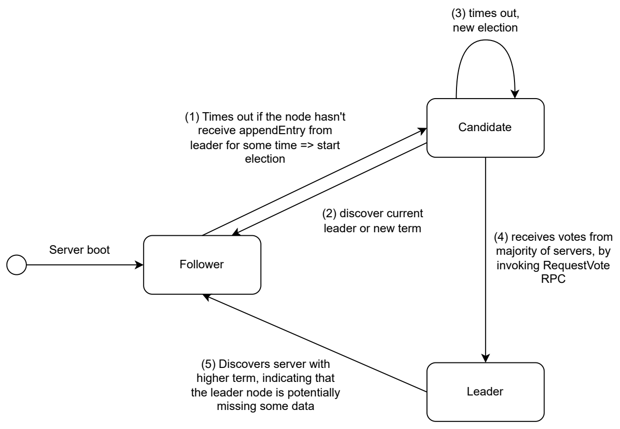
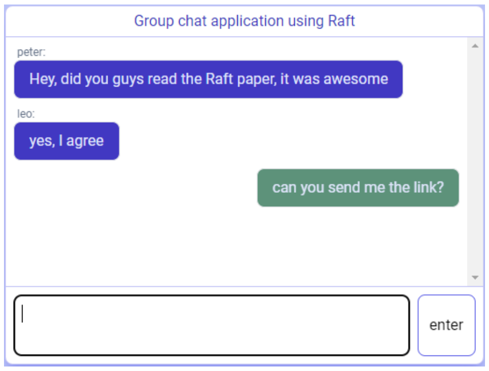

(3) times out, new election
Candidate
(1) Times out if the node hasn't receive appendEntry from leader for some time => start election
(2) discover current leader or new term
Server boot
Follower
(4) receives votes from majority of servers, by invoking RequestVote RPC
(5) Discovers server with higher term, indicating that the leader node is potentially missing some data
Leader

Fig. 2: adapted from [6], state transition diagram for Raft algorithm, indicating the possible role transition



Hey, did you guys read the Raft paper, it was awesome

yes, I agree

can you send me the link?

Fig. 3: screenshot of our group chat application

with a random timeout, for the same reason. Therefore, in expectation even if ”tie” occurs, eventually it will be resolved by the implementation of random timeout.

As we discussed in lecture, a reliable failure detector is impossible in asynchronous networks. Transition (2) ensures that, when a false failure detection occurs, upon receiving the next message from the current leader, the election process will cease while retaining the current leader. Transition (4) happens when the candidate receives majority votes from all the nodes during the election process. Transition (5) is designed for the leader to ”resign”, upon seeing a server with a higher term, and as will be discussed later, a node with a higher term should be elected as the new leader.

Time is divided into terms in Raft algorithm, which increase monotonically over the duration of the application [6]. Generally, a node with a highest term value is concurrent with the most committed logs. Thus, a node with a lower term value should not be elected as the leader since some information might be missing for the node.

B. *Implementation details*

To showcase the diverse application of Raft algorithm, we implemented a group instant chat application (see fig 3). The group chat application is built on a server-client architecture. It allows users with different user id to chat in the same channel, and the other users can see the messages sent quickly. Additionally, we want the message delivery to be reliable, i.e. every message received and acknowledged by the server should be received by the other users eventually. We built the server using Spring boot[1], for the functionality that it provides and the client application using Angular[2], for the powerful renderer that it provides.

The order of messages in our application are both FIFO (first in first out i.e. if a user send message A before message B, every user should see message A before message B) and Totally ordered (if a user see messages in a particular order $A \rightarrow B \rightarrow C$, then all users should see the messages in the same order $A \rightarrow B \rightarrow C$). The total order is ensured by the Raft algorithm, due to the nature of replicated state machine of Raft and the deterministic property discussed in section V-A. For the FIFO order, each message is associated with a user index on the client side, that determines the actual order being sent.

To connect the chat-application client (fig 3) to the service, we declare these endpoints from the server:

1) poll: for client to poll (i.e. fetch) messages from the server periodically to check if there is any new messages.
2) send: deliver a message to server with an unique id that should be received by other users in the same channel eventually.
3) lastMessages: get the last-k (e.g. 30) messages from the servers. Designed for when clients first enter the application they only get the last-k messages instead of the entire list of messages. An infinite scrolling is employed that when users scroll up, the application can continue to fetch older messages using this endpoint also.
4) senderIndex: get the latest userIndex for a user, which is used for FIFO message ordering.

As discussed in section V-A, the replicated state machine is dependent on the application. In our application, our replicated state machine consist of:

1) messageList: a list of messages to store the sequence of messages sent by user.
2) userSequence: a key value map of user id being the key and user index being the value, which is used to ensure FIFO order of messages discussed above.
3) holdingQueues: a data structure for FIFO order, which is used to postpone out-of-order messages.
4) receivedMessages: a set that is used to check and avoid duplicated message.

Each log entry contains

1) uid: to uniquely identify a log entry
2) action: command for the state machine

[1]Spring boot is a versatile backend framework in Java, more details at https://spring.io/projects/spring-boot

[2]Angular is a powerful frontend framework in Javascript, more details at https://angular.io/

3) data: contains the message
4) term: that is used by the Raft algorithm to ensure a larger term node can become the leader.

using the replicated log, all nodes in the system can deterministically construct identical state machines, and therefore provide consistent services for users. Specifically, during each commitment, when the majority of nodes have received a particular log entry, the leader will commit and apply the log followed by its followers. The log contains a create command for a message, that will be appended to messageList mentioned above, in additional to mutating other states.

We also make the service:

1) Highly available: for a system of n nodes, the service can tolerate up to $\lfloor \frac{n-1}{2} \rfloor$ failures; e.g. a system of 5 nodes can tolerate up to 2 node failures.
2) Highly reliable: despite catastrophic failures to the majority of nodes, providing that one node in the system have the full log history, all committed data are intact. As discussed in section V-A, since the node has the highest term value, it will be elected as the leader and replicate its intact log to all other nodes, resulting in a very high reliability.
3) Horizontally scalable: since most of our requests are read request, i.e. the poll requests, having more servers can handle more poll requests, indicating horizontal scalability.

These properties are enabled by implementing the Raft algorithm to distribute the data storage and services across different servers in the network. Accordingly, our service is highly resilient to single point of failure, risk of data loss, resulting in a very robust and fault-tolerant service, creating an illusion of a single service credit to the consistent replicated states machine provided by Raft algorithm.

C. Differences

Despite that our implementation mostly follows the Raft algorithm, there are some slight differences that differentiate our approach with the Raft algorithm that we think will suit our application better than simply adopting Raft algorithm.

In Raft algorithm, servers communicate through RPC (Remote procedure call). The Java equivalent of RPC is RMI (Remote method invocation), which requires a RMI repository to function properly. In order to create a fault tolerant system, the RMI repository needs to also be fault tolerant, which adds complexity to our system model. For our application, we decided that such complication was not necessary and resorted to use http call to Spring boot endpoint instead.

We also decided to forward the message through servers, which is different from the Raft algorithm. In Raft algorithm, when a client first interact with a non-leader service, the service will reject the client's request and assist the client to connect to the leader [6]. We resorted to allowed the clients to connect to different servers to provide horizontal scaling. As discussed above, our requests are read-dominant, having more servers serving the clients' requests can increase the throughput of the system. For this reason, we implemented an approach to let the followers forward the messages from client to the leader, to provide horizontal scalability for the services while maintaining Raft's properties.

D. Critical analysis

Despite Raft provides an easy framework for our application to provide consistent and fault-tolerant service, it also introduces some limitations.

As discussed in section IV-A, one fundamental issue with Raft algorithm is the high latency induced by the log replication process. For an instant chat application, having a lower latency for messages may benefit the users more than having a robust and fault-tolerant one.

Additionally, for a large-scale chat application like WhatsApp, the amount of messages being sent everyday can be tremendously high. Accordingly for our application, the log induced by the high volume of messages can result in a substantially large log file, scaled by the number of nodes in the system since the entire log history is replicated to other nodes.

VI. Conclusion

In this paper, we describe the need for a consensus algorithm for our chat application, and research on several existing consensus algorithms proposed for distributed systems. We analyze and contrast each algorithm and finally choose Raft algorithm to handle the log replication task for our chat application for its simplicity and practical viability. Also, we investigate the practical applications developed based on Raft and the future development direction for our application. Finally, we have a detailed discussion on Raft and its implementation details in our application, and critically analyze some shortcomings of using Raft.

From this project, we learn two key things. One is that the theoretical algorithm might be too abstract to be implemented in the real world application, even though it provides a strong theoretical guarantee on its property. Another thing is that though Raft is relatively simple for study and development, it might not be suitable for all situations, e.g. a latency-sensitive system.

References

[1] G. F. Coulouris, *Distributed systems : concepts and design*. Addison-Wesley, 2012.
[2] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, pp. 133–169, 05 1998. [Online]. Available: https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf
[3] ——, "Fast paxos," *Distributed Computing*, vol. 19, pp. 79–103, 07 2006. [Online]. Available: http://wcl.cs.rpi.edu/pilots/library/papers/consensus/cheap_paxos.pdf
[4] B. Francisco, G. Fabíola, M. Achour, and R. Michel, "Consensus in one communication step," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2127, pp. 42–50, 2001.
[5] L. J. Camargos, R. M. Schmidt, and F. Pedone, "Multicoordinated paxos," 08 2007.
[6] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm (extended version)," 05 2014. [Online]. Available: https://raft.github.io/raft.pdf

[7] T. D. Chandra, R. Griesemer, and J. Redstone, “Paxos made live - an engineering perspective (2006 invited talk),” 01 2007.

[8] F. P. Junqueira, B. C. Reed, and M. Serafini, “Zab: High-performance broadcast for primary-backup systems,” IEEE Xplore, p. 245–256, 06 2011. [Online]. Available: https://ieeexplore.ieee.org/document/5958223

[9] C. McCabe, “Kip-500: Replace zookeeper with a self-managed metadata quorum - apache kafka - apache software foundation,” cwiki.apache.org, 2020. [Online]. Available: https://cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum

[10] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one faulty process,” *Journal of the ACM*, vol. 32, pp. 374–382, 04 1985. [Online]. Available: https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf

[11] “Quorum queues — rabbitmq,” Rabbitmq.com, 2024. [Online]. Available: https://www.rabbitmq.com/docs/quorum-queues

[12] F. B. Schneider, “Implementing fault-tolerant services using the state machine approach: a tutorial,” *ACM Computing Surveys*, vol. 22, pp. 299–319, 12 1990.

[13] “Kraft overview — confluent documentation,” docs.confluent.io. [Online]. Available: https://docs.confluent.io/platform/current/kafka-metadata/kraft.html