Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Improving RAFT

Semester Thesis

Christian Fluri

fluric@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Georg Bachmeier, Darya Melnyk
Prof. Dr. Roger Wattenhofer

July 3, 2017

Acknowledgements

I thank my two supervising tutors Georg Bachmeier and Darya Melnyk who guided me capably through this semester thesis. We met on a weekly basis where they assisted me helpfully whenever problems appeared. They also supported me actively during the writing process and made sure that the academic standards were met.

Abstract

In this semester thesis, the goal was to fully understand the RAFT algorithm and then to examine its functionality and robustness in a critical point of view. With our own implementation of RAFT in Python we tried to get more familiar with the algorithm and also got a tool to test the algorithm under many different conditions. Especially we tried to measure how well it works in the following two configurations:

- An isolated server: It is assumed that one server still can send messages but does not receive messages from others anymore
- Partition: Here the servers are divided into two subgroups where between these two parts no communication is possible at all.

For the isolated server there were some severe problems which were efficiently solved by some small adjustments in the algorithm.
For the partition there were some limitations visible concerning the scalability in the leader election. We could improve the performance and loosen the limitations by dynamically adapting the election timeouts. Additionally, we proposed a multi-level RAFT and argued why it might be a better solution than trying to improve the leader election further.

Contents

Chapter 1

Introduction

1.1 Introduction to RAFT

Since 1989 there exists a non-byzantine, asynchronous consensus algorithm for managing a replicated log which is called Paxos [1]. The author, Leslie Lamport, submitted a paper which explained an algorithm with the aid of an analogy to an old fictional Greek tribe on an island Paxos. Although the analogy makes sense, the readers seemed to miss the essential meaning of the colourful story and rejected the paper first. At 1998 Lamport handed in the paper again and it finally got accepted with great success. However, it still looked like the nice Paxos story confused some of the readers and he published a simplied version [2] three years later.
RAFT[3], which claims to be an equivalent but simpler algorithm, was developed on the basis of the Paxos algorithm with the goal to significantly improve its understandability. It divides the consensus problem into three sub-problems Leader election, Log replication and Safety.

- **Leader election**: First, from all the servers a leader has to be elected that will be responsible for the replication of all the so called log entries across the server cluster. The servers have to know the total number of servers in the cluster in advance. Otherwise, the servers cannot recognize how many votes they need for a successful election.
- **Log replication**: The leader will receive commands from a client and replicate them to all the other servers. Whenever the leader distributes such a log entry to a majority of servers, the command in the entry is called committed and it is made sure that it will not be removed anymore.
- **Safety**: To ensure the safety and consistency of the consensus algorithm there must hold some properties in the leader election and the log replication so that committed commands eventually will be executed at all the servers and will not be manipulated anymore.

Leader election

A server can be in one of the three states, namely *Follower*, *Candidate* and *Leader*. At the very beginning each server starts as Follower. Each server waits until it either receives a 'heartbeat' message from a Leader, or too much time since it received the last heartbeat elapsed; in the latter case, we speak of a 'timeout'.
As there are only Followers at the beginning, some servers eventually will reach a timeout, which is set uniformly at random within certain boundaries. A server that has reached its timeout tries to become the Leader and therefore transforms to the Candidate state.
A leader election starts with such a Candidate asking each server for its vote and all servers give their vote to the first valid Candidate which requests it. A Candidate becomes a Leader as soon as it gets a quorum of the votes, which must be more than half of all the possible votes, and the election ends. To update every server about the end of the election and to prevent new timeouts, the Leader starts to repeatedly send a heartbeat to all other servers. Some other potential Candidate will then return to the Follower state when it receives such a heartbeat from a current Leader.
However, several servers might timeout at the same time and none of them gets the quorum of votes (e.g. two Candidates could get each exactly half of the votes). This so called split vote ends up in a new election round where each Candidate starts to request for votes again after a randomly set timeout.
Each server divides time into terms. As the whole environment is asynchronous, two server do not have to have an equal term number at the same global time. To maintain a certain synchronisation, every message contains the current term number of the sending server. Whenever a server receives a message with a newer term, it will update its own term to the newer one and become a Follower, if it was not already a Follower. On the other hand, a server replies to a stale message with a negative answer and its current term to update the stale server. In the case of a timeout, a Candidate will always first increment its term number and claim to be the Leader of this new term (I.e. there is a Leader in term 1 and one of its Followers reaches a timeout. Then the new Candidate will start a leader election in term 2 and when receiving the VoteRequest, all the other servers update their terms to 2 as well. This also includes the previous Leader which moves to Follower state for term 2). This way there will be only one Leader possible in a given term and every node can be sure about the Leader for the rest of the term.

Log replication

After a Leader is elected for a given term, the log replication can begin. Whenever a client sends a command to a server, this server will redirect it to the

current Leader, if such is known. The Leader will append the command to its own log and try to replicate this log entry to all its Followers. As soon as half of the servers have appended this command and answered positively to the Leader, the command will be called committed. Now the Leader will apply the command to its state machine and tell the client that this command eventually will be executed on all servers.
Without crashes this already works fine but each of the servers, the Leader and its Followers, could crash at any time. In this way it can happen that some servers have missed some committed log entries or that they have an arbitrary number of uncommitted log entries and these logs have to be restored somehow. This problem is solved by comparing the Leader's log with the Follower's one. If there are differences, then always the Leader's log is right - it will never manipulate its existing log entries - and the Follower has to adapt its log. Accordingly, the Follower always has to check first if its previous log entries coincide with the Leader before it appends a new entry from the Leader. If this is not the case, the Follower will delete all entries up to the most recent coinciding one and then stepwise ask for the missing messages and append them. Continuing like this, the Followers will eventually have the same log entries as the Leader. Every log entry contains the term number at which it was appended to the Leader's log and an increasing index number to identify its positions.
However, with these rules the Leader could still have missed a committed command (i.e. by being down for some time while another Leader committed a new command) and could remove it from all the servers. Therefore another restriction is given in the next Subsection, to ensure that a potential Leader already contains all previously committed commands.

Safety

To maintain consistency of the committed commands among the servers and clients there has to be one more elementary rule in the leader election. A server which does not contain all the committed log entries, has to be prevented from winning an election, otherwise this server can delete some of the committed entries again. For that the Followers only vote for Candidates which are at least as up to date as themselves. At least up to date in this case means, that the last log entry of the Candidate has a higher term as the one of the Follower or the last log entry of the Candidate has the same term and an index at least as high as the last log entry of the Follower. This way only the Candidates with all committed log entries have a chance to win an election since committed entries are already present in the majority of the servers and therefore are required in the Candidate's log for a quorum of the votes.
With these restrictions it can be shown that none of the committed entries will be deleted again and eventually all the committed entries will be executed on every server. In addition the following properties from the RAFT paper [3] hold:

- If two entries in different logs have the same index and term, then they store the same command.
- If two entries in different logs have the same index and term, then the logs are identical in all preceding entries.

Messages

For the RAFT implementation only two kinds of messages are required. These are:

- RequestVote: This is the message sent by a Candidate to start an election and to ask every server for its vote. The reply, which contains a positive or negative vote for the Candidate, will be the RequestVoteReply.
- AppendEntry: This message is responsible for the log replication between the Leader and its Followers. Whenever some log entry has to be added at the Follower, the Leader sends an AppendEntry message which contains one log entry. The Follower will then use an AppendEntriesReply message to give a feedback to the Leader. When there is no more log entry to be added at a Follower, the Leader sends an empty AppendEntry message which is used for the previously mentioned heartbeat.

For further details the interested reader is referred to the following website[1].

[1] `https://raft.github.io/#implementations`

Chapter 2

Implementation

To simulate and really understand the RAFT algorithm we implemented it by ourselves. Therefore we needed a suitable programming language and a useful IDE (Integrated Development Environment). Another important requirement was that the message passing between servers and clients is fast and reliable enough. For this we had to choose a powerful asynchronous messaging library and a threading library for a fair scheduling among the server processes. Having the communication working, we could start to structure the code and implement the logic for the algorithm with all the additional features for the simulation.
For this semester thesis we did run the whole simulation on one machine. Therefore we created all server processes as independent threads and let them communicate via sockets.

2.1 Programming environment

As programming language we chose Python 3.6[1]. The available threading library[2] with a fairly distributed scheduling seemed to be quite promising, what we need for a realistic distributed simulation like with RAFT.
As IDE we selected Pycharm[3] which makes possible a good handling of the code. Additionally, the Anaconda 4.3.1 package[4] makes it convenient to run Python and Pycharm on Windows.
The library for the asynchronous messaging is ZeroMQ[5] which was suggested to us because of its handiness and large, strong functionality.

[1] https://www.python.org
[2] https://docs.python.org/3/library/threading.html
[3] https://www.jetbrains.com/pycharm/
[4] https://docs.continuum.io/
[5] http://zeromq.org/

2.2 Message passing

The first important thing for a fast and realistic implementation was the message passing. It had to be made sure that the messages were sent and received in a fairly distributed way and in as little time as possible. Therefore, we first implemented a naive function that repeatedly tried to asynchronously receive messages in a round robin fashion. After realising that this first approach did not scale for larger numbers of servers, it came to an essential second approach where we used socket listeners to receive the messages.

2.2.1 Asynchronous read with Round Robin

In RAFT every server has a direct link to each other server and to each client. Accordingly, each server has to listen for messages from each other server and client. A normal blocking socket-read would have caused that a server (as one thread) could only listen to one other server at a time. In this case it could have happened that it waited a long time for one message of a server while the other servers may already have sent many other messages. This would not have been a good model for concurrency indeed. As a consequence we made the socket-read non-blocking. This way the server asks a socket for a message and immediately returns if there is no message available. Every server now continuously checks all its sockets in a round robin fashion for new messages and handles it accordingly. This approach guarantees fairness from the socket requests in round robin and a roughly realistic concurrency from the thread scheduling.
However, it turned out that those non-blocking reads still took too much time and the simulation could not keep up with the message handling of a larger amount of servers. Therefore the messages were received more and more delayed and the simulation was not really representative anymore.

2.2.2 Socket listener

As the previous procedure did not scale nicely, we tried to get rid of the time consuming non-blocking reads and placed a socket listener in front of each socket. This way for each socket listener has to be a new thread generated that constantly performs a blocking socket-read. Whenever it receives some message it restarts the reading process and places the received message in a queue. The only thing left for the server is to listen for messages in the queue in a FIFO (First In First Out) way.

To test the performance of those two approaches, we set up a simple message passing test. In the setup a message is sent from one server to another in a round robin fashion, i.e. server 0 sends a message to server 1, then server 1 the same message to server 2 and so on until the last server sends it again to server

0. Then we measured the time for several rounds and took the average time to send a message from one server to another.
This test should roughly imitate the communication part of the RAFT simulation where only a few sockets receive a message and most of the sockets hear nothing at some time instance. This way we hope to get a good estimation of the communication effort for the two approaches.

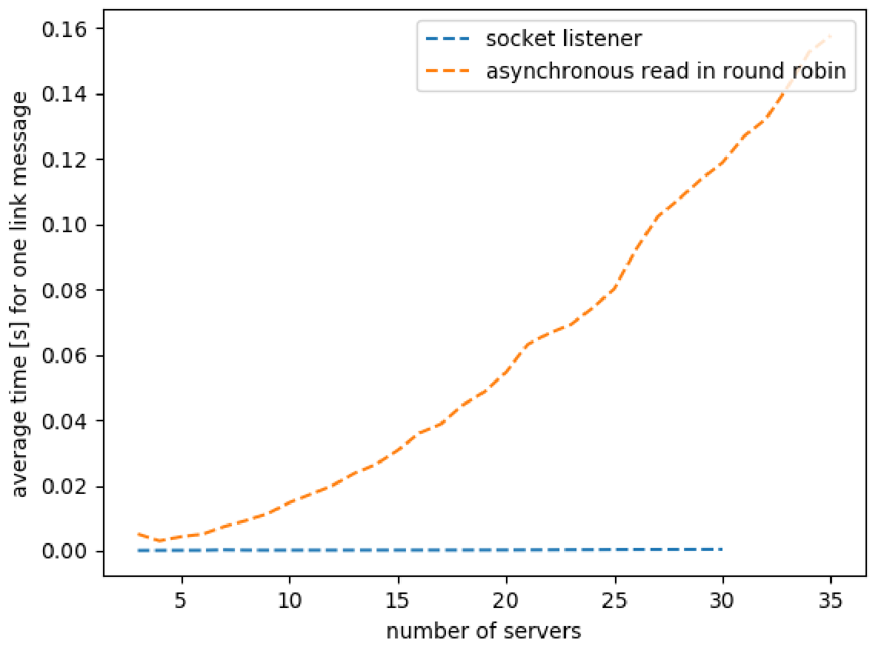

Figure 2.1: Comparison between the two approaches with the message passing test

We ran the test with a different number of servers for the two communication approaches. In Figure 2.1 we see that the asynchronous read is much slower while the socket listener is much faster and scales better. It seems to be preferable to have more threads than having a lot of non-blocking reads. As a result, we used the second approach. Still, there is a linear increase for the socket listener from 0.165 ms with 2 servers to 0.5 ms with 30 servers which is not visible from Figure 2.1.

2.2.3 Threading

For the threading, we used the standard library Threading. To provide concurrency in this distributed system, threads are essential. For every server there is one main thread which reads from the message queue and reacts according to the messages, i.e. it sends a reply to the Leader server. Additionally, there is one thread for each socket and a thread for each timeout that is needed to tell when the timeout happens, i.e. the election timeout. For the clients it works equivalently. The threading scheduling should then provide some pseudo concurrency.

2.3 Code structure

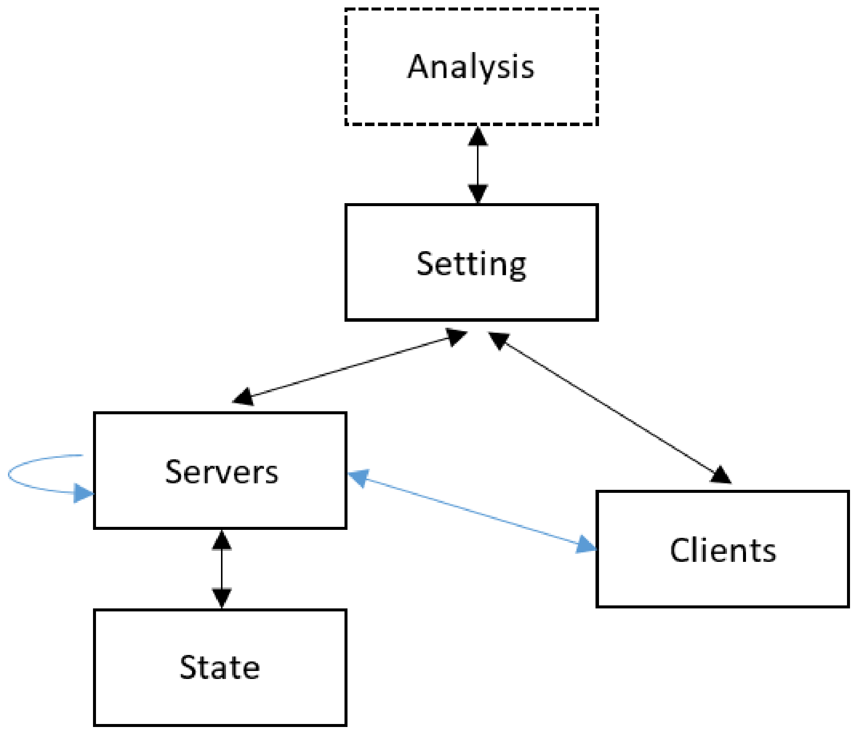

Figure 2.2: The code structure for the RAFT implementation with its main objects

Figure 2.2 shows the hierarchy of the code implementation. On the top is Analysis which builds the interface to the user. From there different simulations with the according parameters can be run. As the user starts such a simulation from Analysis, an object Setting is created. Setting however creates the right number of server and client objects. It is responsible for starting the whole simulation and also for closing it correctly again. Server and Client both do the timeout handling and the socket communication for RAFT which is indicated by the blue arrows. For the RAFT logic of the servers there is an additional object created from Server, called State, which handles every received message

of a server.
To provide some order, changes in the singular objects are generally done along the black arrows. I.e Analysis should not change a link failure in Server directly, it tells Setting to do this manipulation in Server.

Chapter 3

Interpretation

In this chapter we analyse the implementation of RAFT on different configurations. Therefore some kind of measurement is essential. This is done by two features:

- Global time: Every server and client takes the global start time as soon as it starts running. Whenever one of them has a termination condition fulfilled, it will take the global end time and the difference of end and start time will be given as output.
- Term: Every server maintains its own term, starting from 1, which is either increased by a timeout of the server or by a message with a newer term. It can be used as indication of how stable a Leader is, i.e. if it sends heartbeats to its Followers reliably enough, and of how effective the leader election is. The output will be the term number of the corresponding server at the end.

The termination condition is either a successful leader election where a server becomes Leader or a predefined number of committed commands in the client, depending on what we want to measure.
There are many cases where the time and the term strongly correlate and it does not make sense to look at both. Therefore, most of the Figures only show the global time.
In the following sections we will take some measurements, try to improve the performance of RAFT and also add some new functionalities to the original algorithm, such as sort of a sliding window or dynamically replicated messages to deal with failures. We analyse the two special cases of an isolated server and the partition of the servers where we spot some issues. Corresponding to these problems, we propose and test some new mechanisms and finally, the results will be further discussed and the idea of multi-level RAFT presented.

3.1 General

First, the algorithm is analysed with non-negligible failures between the servers. Therefore we compare the two policies Replica VoteRequest and Replica AppendEntry with the original implementation.
Secondly, we extend the RAFT algorithm with a sliding window mechanism which was already proposed in the paper as extension.

3.1.1 Replica VoteRequest

It might be that the links between some servers have a non-negligible failure rate. In such an environment many leader elections fail due to the lack of received VoteRequests and VoteRequestReplies until a Candidate gets lucky and wins the election. The original algorithm has no mechanism to effectively react to it and thereby the whole progress is significantly slowed down the higher the failure rates get. As there are two successfully received messages needed for one vote, these are a VoteRequest and the according VoteRequestRely, we even expect a non-linear increase of the required terms for a leader election with increasing failure rates.
To dynamically deal with failures we built a naive policy, called Replica VoteRequest. It means that the VoteRequest (and its reply) messages are sent several times. The number of times is equal to the number of terms that went by since the last known Leader was active. I.e. if the Candidate starts an election in term 4 and its last known Leader was at term 1 active, then it sends each message 3 times. In this way at the beginning some elections may still fail but after some terms the messages should be sent so many times that at least one message reaches its target with a high probability.

In Figure 3.1 we can see that without Replica VoteRequest the number of required terms increases non-linearly the higher the general link failure gets. However, with replica VoteRequest the number of terms remains considerably low. As a consequence we can conclude that even a quite naive mechanism like Replica VoteRequest improves the leader election a lot.

3.1.2 Replica AppendEntry and Sliding Window

After a server could successfully establish itself as Leader, the same problem arises as before. To prevent timeouts, each Follower has to more or less reliably receive heartbeats from the Leader. But the higher the link failure gets, the less heartbeats will reach the Followers and eventually some timeout appears which starts a new leader election and the whole progress gets stuck again.
Similar to the Replica VoteRequest we implemented a more adaptive Replica

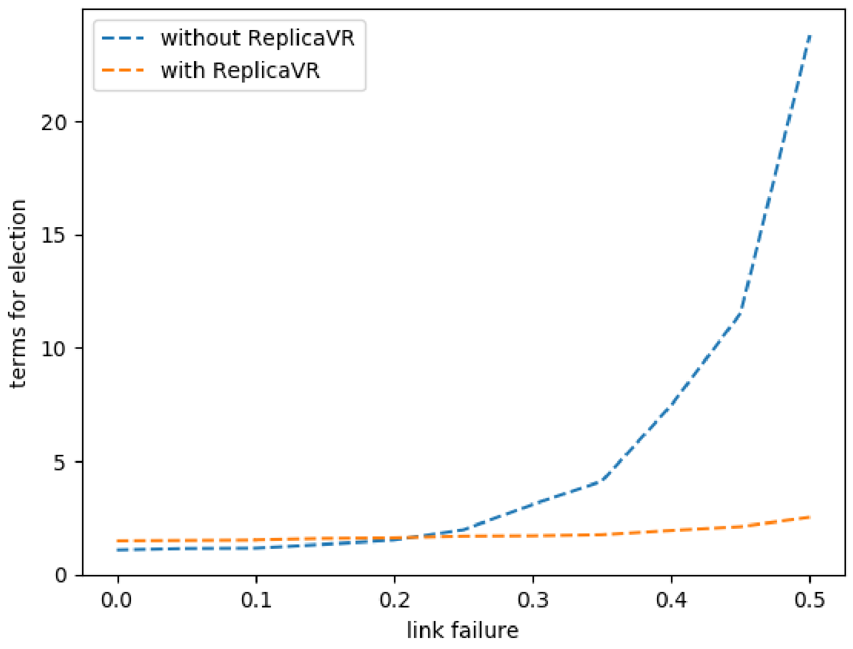

Figure 3.1: The impact of the Replica VoteRequest on an election with non-neglicible link failures (timeouts: 0.1-0.15[s], servers: 10)

AppendEntry policy. The Leader sends, as normal, every heartbeat new AppendEntry messages to each Follower. It then remembers at each time if it received a reply in this heartbeat period from the specific Follower. If it receives a reply, then it subtracts 0.1 from a factor, which corresponds to the number (the floor rounding of it) of times the AppendEntry is sent to this Follower. If it does not receive any reply in this period from this Follower, it doubles the factor and therefore sends twice as many messages at the next heartbeat. We chose this mechanism similar to TCP with a multiplicative increase and an additive decrease.

Another important concept is the sliding window mechanism. In the original algorithm the Leader can only send one command at a time with an AppendEntry and then has to wait for an AppendEntryReply or for the next heartbeat timeout before it is able to send the next command. With the sliding window the Leader can send several different AppendEntries at once, if there are more commands left to be sent. It does not have to wait for an AppendEntryReply which saves some time. However, it happens that a newer AppendEntry overtakes an older one in the network or that some messages get lost. To preserve the correctness of the algorithm, these non-consecutive commands cannot be added

to the log until the missing commands in between are received as well. Therefore we needed some buffers for the Followers, to store the commands that were not expected yet.
We also implemented a sliding window mechanism between client and Leader which is described in the Appendix A.1.

In Figure 3.2 we simulated 5 servers and measured the time and number of terms until 100 commands were committed. Therefore we wanted to see what impact these two mechanisms have.

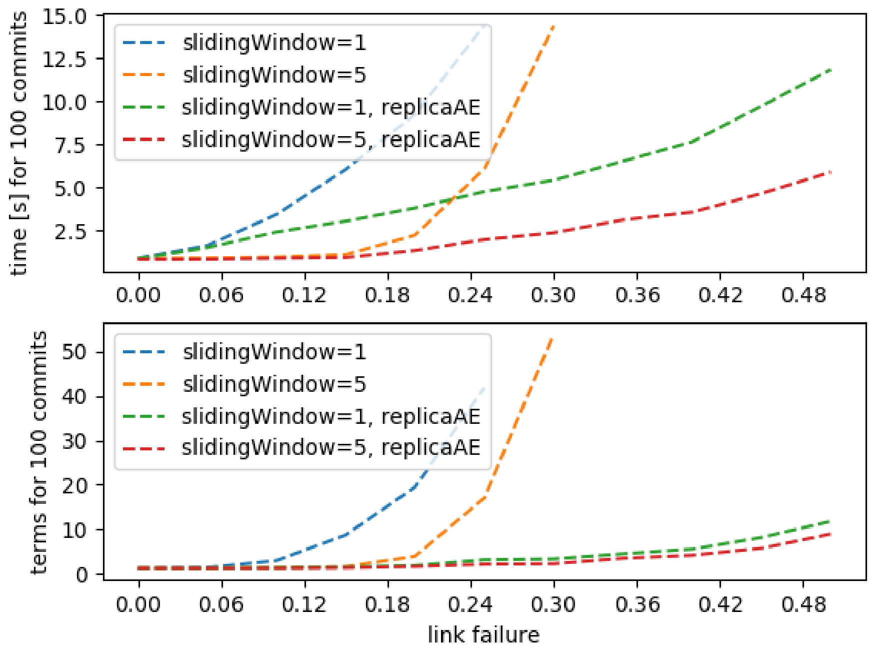

Figure 3.2: The impact of the Replica AppendEntry and the sliding window size on the RAFT progress (timeouts: 0.2-0.3[s], heartbeat: 0.1[s], servers: 5)

When looking at the number of required terms in the Figure, it becomes clear that for higher link failures much less terms are needed with Replica AppendEntry. This mechanism prevents timeouts of the Followers, as intended, and thereby less elections are needed and the progress becomes faster.
On the other hand the sliding window generally accelerates the progress which is visible by comparing the red and green curve of the time plot, although it does not really affect the number of terms. This comes from the fact that there are more messages sent in the same amount of time which obviously does not have an impact on the number of required terms. Additionally, the sliding window

without Replica AppendEntry seems to prevent new leader elections in some degree. This might be because of the higher amount of received messages with a sliding window which also prevents some timeouts.

3.2 Isolated server

It could happen that a server suddenly has a problem and can not receive any messages from the other servers anymore. Although, it is assumed that the isolated server still can send messages to the other servers. We now consider different cases and find weaknesses of the RAFT algorithm where i.e. delays appear or even the whole progress of the other servers is stopped. To these problems we represent some alternative mechanisms.

3.2.1 Isolated server is Leader

First we assume that the isolated server starts as Leader. The Leader still can append new entries to the other server's log but it will never know if its Followers could successfully append it. Without feedback the Leader cannot commit any new commands and thereby is unable to make any progress. Unfortunately, a Leader will not give up its state voluntarily. Especially in conditions with negligible link failures, it will be a serious problem as the Followers always receive a heartbeat and therefore will never have a reason to start a new election. As a result, the original RAFT algorithm will stay in this state forever.
To solve this problem we have added two new mechanisms:

- AppendEntryReply Timeout: The leader has a new timeout which will go off as soon as it does not receive a AppendEntryReply for a certain time period. If this happens, the Leader knows that something goes wrong as it should receive some replies from its heartbeats. As a consequence it will reset itself to the Candidate state and try to renew its leadership.
  As this server is still isolated, it will not receive any VoteRequestReplies and will not be able to become Leader again. Eventually, a new leader will be elected and the non-isolated servers become able to make progress again. While this mechanism suffices for this kind of configuration, there could be two or more servers isolated which still get messages from each other. In this case a potential isolated Leader will still receive a reply from the other isolated servers, the AppendEntryReply Timeout will never occur and RAFT stays blocked.
  Therefore we needed another mechanism.
- Commit Timeout: We add an additional timeout to the Leader which will go off after a certain time without a new commit elapsed (if there is still some command to be committed). Whenever this timeout happens,

it indicates that the Leader cannot access to a majority of the servers anymore since for a committed command this is the requirement. This Leader becomes useless and stops the whole progress. Therefore the Commit Timeout will force the Leader to give up its leadership and to restart as Candidate.
In the case of an isolated Leader, including a minority of other isolated Followers, the Leader will always be reset to Candidate state after some time and thereby enable the non-isolated servers to find a new Leader. On the other hand a successful Leader will rarely be reset as it is able to commit new entries with high probability. Therefore the Commit Timeout does not interfere with normal configurations which make progress.

3.2.2 Isolated server is Candidate

With the two previously mentioned mechanisms the isolated server might be prevented from being a Leader, however it still can interrupt the progress in the Candidate state. As the isolated server does not receive any AppendEntry messages from the others, it continues via timeout to increase its term and repeatedly starts new elections. Therefore it keeps on interrupting the progress of the others because it forces them to update their term and the Leader has to restart as Follower again. As a consequence the others have to find a new Leader before the isolated server timeouts again. After they have found a new Leader, they can only continue until the isolated server reaches a higher term again and it goes on.
One solution to prevent interruptions by an isolated server is the so called LastLeaderTerm Policy. To the already existing safety requirements in the Leader election, that the last log entry of the Candidate has to be at least as up to date, a new requirement is added and checked first. Each RequestVote has to contain the LastLeaderTerm which is the last term the Candidate has seen a Leader. The server that receives this RequestVote message first checks if its own LastLeaderTerm is higher. If this is the case, the server will reject this message by just sending a negative reply and it exceptionally will not update its term. If the LastLeaderTerm of the Candidate is not higher, the Follower proceeds this VoteRequest as normal.
Unfortunately, this change does not grant the safety of RAFT anymore which is shown in the following example:

- There are 10 servers, 6 of them got A as the last log entry and 4 of them got an additionally last log entry, after A, called B. It can happen that one of the servers with last log entry A becomes Leader but is only able to send a heartbeat to the servers with last log A before it crashes again. Now the servers with last log A have a higher LastLeaderTerm then the ones with B and will not vote for a server with B. On the other hand the servers with B will not vote for a A server either as the last log is less up to date. From now on no server will be able to reach a quorum of votes and it is shown that even one server could stop the whole progress.

Accordingly, we need to adjust the LastLeaderTerm Policy. Therefore the policy is restricted to the current Leader as follows:

- When a Follower still receives heartbeats from a current Leader (current term is equal LastLeaderTerm) then it only accepts VoteRequests of Candidates which have already seen this current Leader (they will have the same LastLeaderTerm) else it just rejects the vote again and does not update its term.
  Additionally, the Candidate will reset itself to Follower state and its term to the received LastLeaderTerm whenever it receives a VoteRequestReply which contains a higher LastLeaderTerm. In this way we enable that this Candidate can possibly listen to this newer Leader again, else it would be lost since it will have a higher and higher term.

With this change there will not be any kind of deadlock anymore. With the Commit Timeout the Leader already gives up its leadership by itself after not having committed anything for a while, otherwise it still makes progress. When there is no more Leader, then the LastLeaderTerm Policy has no effect on the Leader election and it continues as normal where the safety properties are already proofed in [3].

3.3 Partition

In a network it can also happen that some part of the servers get completely cut off from the rest and cannot communicate with the other part at all. One simple example is given in Figure 3.3.

As seen in the Figure 3.3, from now the servers 1 to 3 can only communicate to each other and server 4 only to server 5. In this case there will run two separate leader elections simultaneously. However, only on the side with the majority of the servers it can be successful. Since the servers know the total number of servers and positive votes from a majority of server are required to become Leader, they need 3 out of 5 votes. As the smaller side only consists of 2 servers, it is impossible to get a Leader there.

In the following Subsections the behaviour of RAFT in partitions is analysed. One important concept, namely the 'fake' partition, is discussed to test the reliability of the simulation. The partitions are tested with different timeouts and with the existence of fixed failures and delay which will be explained in more detail.

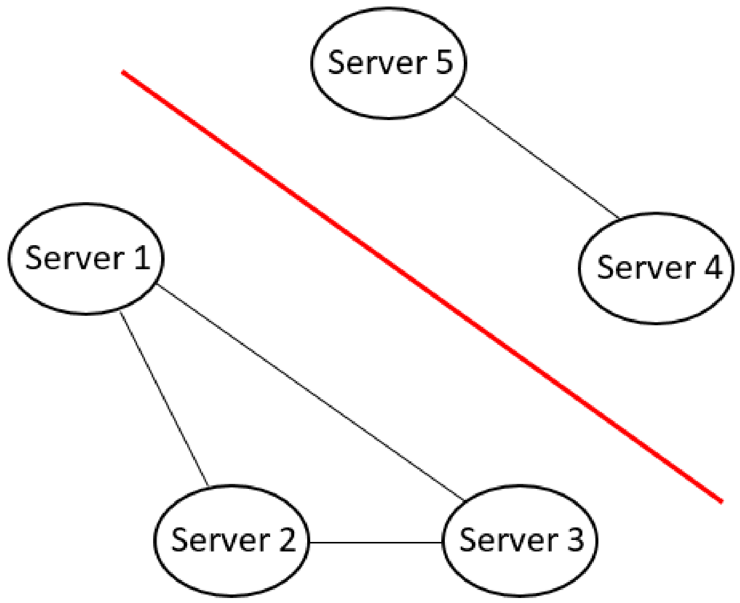

Figure 3.3: Example of a partition with 5 servers where 2 servers get cut off

3.3.1 Reliability of the simulation

When partitions are tested, one important feature is the time for a successful election. We can fix a partition ratio, i.e. a third of the servers is cut off from the other two thirds, and then analysed how the time for a successful leader election evolves versus the total number of servers. On the other hand we can also fix the total number of servers and inspect the election times for different partition ratios. This sounds quite simple however it is not guaranteed that the simulation works reliably, especially with a higher amount of servers. In example an increase in the time curve could be caused by the partition configuration itself, what we wish to analyse, or then by the more intense effort of the simulation caused by more servers. Ergo, we have to interpret the results carefully and first need to check if we can believe in our simulation.
To get a hint of how reliable this simulation is, we run a so called 'fake' partition. Here we ignore the minority that got cut off from the partition and only simulate the part of the partition which can build a majority and therefore is able to find a Leader. The other part of servers which is left out, should not have an effect on the leader election time as it is redundant for the progress. For the example in Figure 3.3 only the servers 1 to 3 would be simulated as they can still get enough votes from each other. The servers 4 and 5 will be left out because their leader elections will never be successful and only consume some calculation time of the simulation.
In principle the results of the 'fake' partition and the real partition, with all

servers simulated, should coincide. If they do, we know that the performance of the algorithm does not depend on the simulation effort and that we can rely on these measurements.

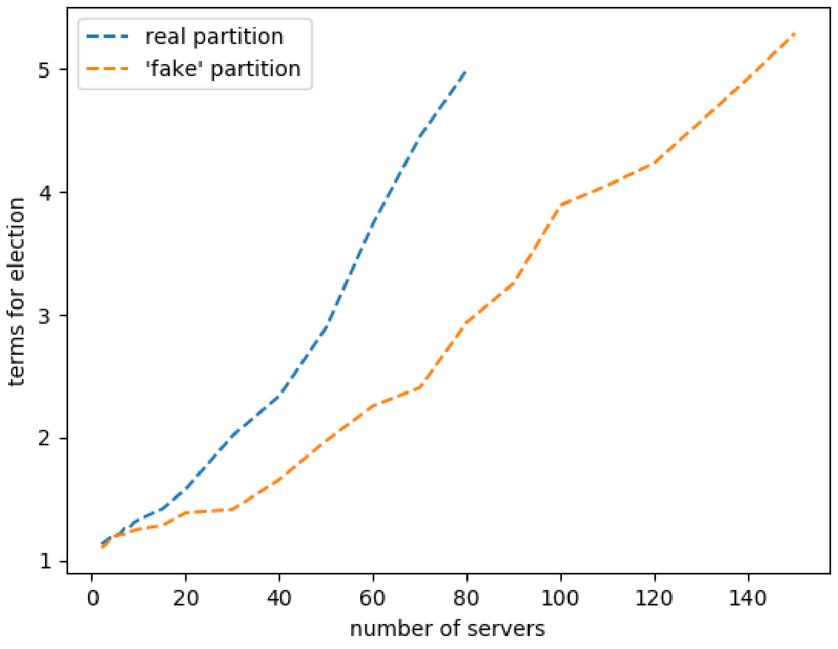

Figure 3.4: Comparison between the results of real partition and ’fake’ partition with a half-half partition (such that there is still a majority - for 100 servers the partition would be 49-51) and timeouts: 1.0-1.8[s]

In Figure 3.4 it is visible that for a higher number of servers, the effects of the more involved simulation are not negligible anymore. The real partition even gets stuck after 80 servers due to overloaded simulation while the ’fake’ partition is able to continue until 150 servers. As a result, we concentrate on a smaller amount of servers and hope to get a better reliability there.

In Figure 3.5 we see a much better correlation between the two kinds of simulation. Here the total number of servers is fixed to 10 and the number of servers, that were cut off, varies. The election times nearly coincide except for the case where 4 servers are cut off. But even latter case only deviates by a smaller factor. We therefore restrict our simulations to a smaller amount of servers.
Additionally, we see from Figure 3.5 that the election time increases non-linearly the more servers are cut off. This might come from the fact that for a larger

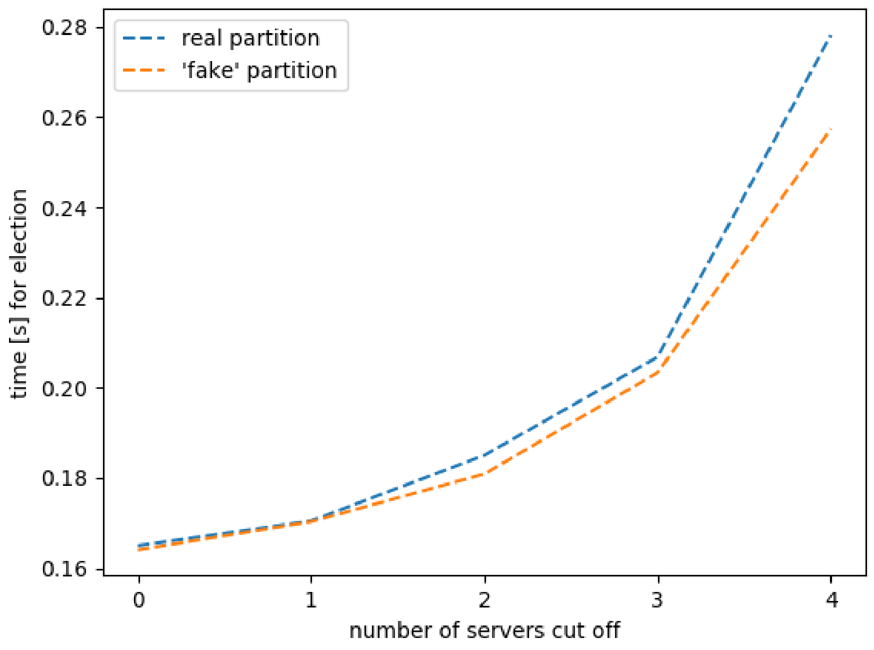

Figure 3.5: Comparison between the results of real partition and 'fake' partition with a different number of servers cut off (timeouts: 0.1-0.15[s] and 10 servers)

partition a higher percentage of the other servers has to answer positively to the VoteRequest. Therefore the cases where two or more servers timeout roughly at the same get less successful as they steal each others votes.

3.3.2 Different timeouts and policies

As we are restricted to a smaller amount of servers, we cannot exactly predict how the algorithm behaves with more servers which might be even the more interesting part. Therefore we tried another way to imitate the conditions with more servers. Instead of increasing the number of servers we can just make the lower and upper timeout bounds tighter. I.e. if we halve the timeout interval then the density of timeouts will be doubled; as if the timeout interval remains like this and the number of servers gets doubled.

In Figure 3.6 we again simulated 10 servers with a varying number of servers that were cut off for different upper timeout bounds. From these curves we see that the election time continuously increases as the timeout interval gets halved. At a certain time interval the election time seems to increase even more and the algorithm will start to become unstable at a certain point. This can be explained

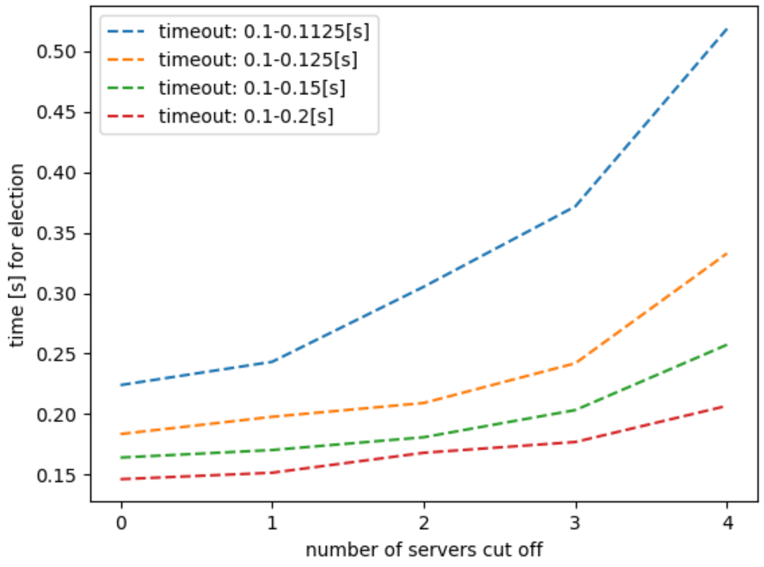

Figure 3.6: Comparison between different timeout intervals (10 servers and 'fake' partition)

by the increasing timeout density, when the timeout bounds get tighter.
As a consequence the number of servers seems to be limited for a fixed timeout interval. When the number of servers gets too high, the upper timeout boundary needs to be increased. RAFT has no dynamic timeout adjustment yet and therefore most certainly will not work fluently for a drastically increased number of servers.
To deal with this issue, we introduce two different approaches:

- increaseTimeout: Each server remembers the last leader term and counts the consecutive terms without Leader. Every time there is a new term without leader, it increases its upper bound of the timeout by the original interval length of the timeout (upper bound minus lower bound). In this way the interval increases linearly until there is a new leader found.
  This can be done as well with an exponential increase of the boundary.

- increaseCandidateTimeout: The Candidate counts the number of positive and negative votes it got. Then it adapts its upper timeout boundary for the next timeout according to the ratio between the positive and negative replies. Therefore, Candidates with many negative votes get handicapped in the next timeout by a larger timeout interval. On the other hand, the Candidates with a lot of positive votes get a tighter timeout interval. In this way we want to give the more promising Candidates an advantage.

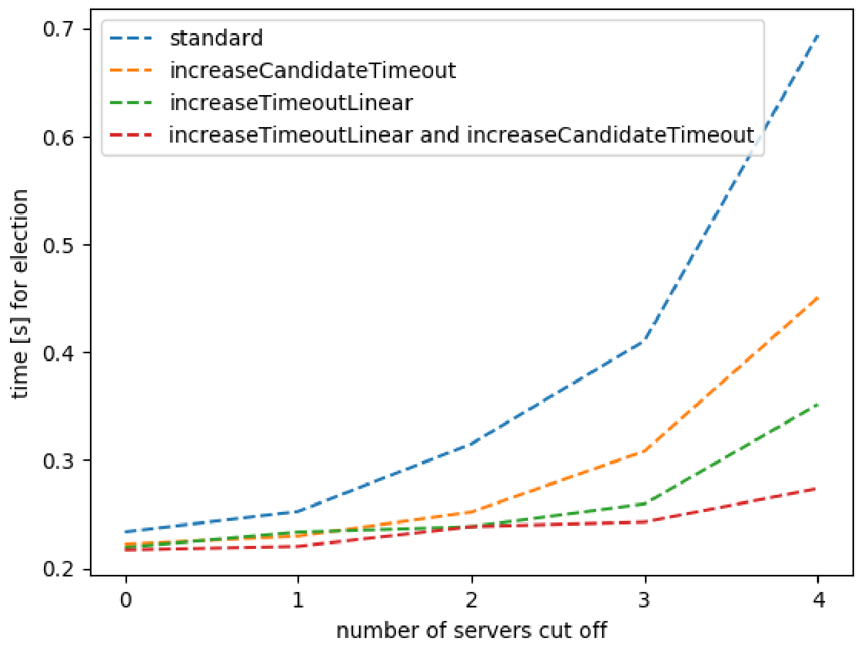

Figure 3.7: Comparison between the different timeout policies (timeouts: 0.1-0.10625[s], 10 servers, 'fake' partition)

In the Figure 3.7 our goal was to compare the performance of RAFT with the newly introduced policies. We used 10 servers and a tight timeout boundary of 0.1-0.10625[s] where we varied the partition size. The curves show a significant impact of the two policies. Both policies have a positive effect on the performance whenever a larger percentage of the servers gets cut off. Even the two policies together seem to improve the progress more which indicates that the policies are not making the same improvements.
We also tested the so called increaseTimeoutExp with an exponential increase of the upper timeout boundary but it is not that effective in this configuration as the increaseTimeoutLinear. This might come from the fact that the timeout interval is not tight enough and an exponential increase has a too rough adjustment and is not necessary here. This might change for cases where the number of servers is by magnitudes higher for this timeout interval which remains to others to test.

3.4 Further observations

In this Section we introduce the concepts of fixed failures and fixed delays which imitate some problems that can appear in networks like the Internet. Then we add these two features to the RAFT implementation and shortly discuss their impact based on some measurements.

In the Internet it happens that a direct link between two servers fails because a router in between has crashed or a link is slower due to more traffic i.e. To simulate such behaviours, we added two parameters to the implementation which will be set right at the beginning of the run time:

- fixed failure: At the beginning, each directed link - a server has two links to another server, one for sending and one for receiving - between two servers will be disabled with a given probability for the rest of the simulation. I.e. with the given fixed failure of 0.1, on average each tenth link will be destroyed and will not work during the simulation.
- fixed delay: For all directed links a fixed delay will be chosen uniformly at random in a certain delay interval. For the whole simulation the messages through this link will have this fixed delay, additional to the actual receiving time of the sockets.

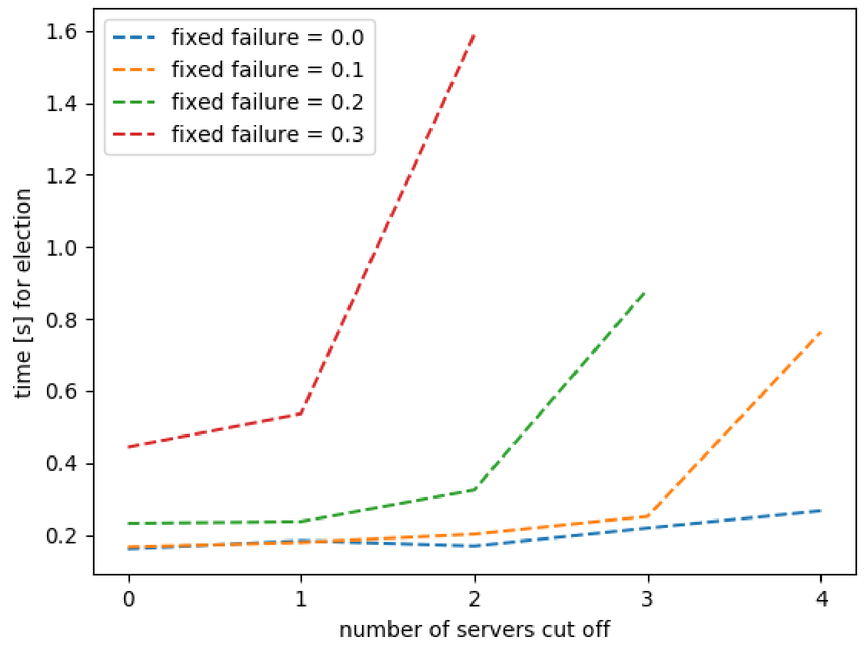

Figure 3.8: Comparison between the different fixed failures and partitions (10 servers, timeouts:0.1-0.15[s], 'fake' partition)

We wanted to see how much impact link failures have on a partition of servers. In Figure 3.8 we simulated leader elections with 10 servers and varying partition sizes for different fixed failures. From the curves we see that a higher fixed failure together with a large partition becomes very critical. This can be explained by the fact that in large partitions a Candidate nearly needs all of the votes of the servers in its partition. When now some links fail between this Candidate and other servers, it needs an even higher fraction of the available votes or it might happen that it cannot reach a majority of the servers anymore. I.e. from the 10 servers, the servers 7 to 10 are cut off. Additionally, there is a link failure between server 1 and 2. Then servers 1 and 2 will never be able to become leader. However, they will also timeout and interfere with other Candidates which makes the election harder for the others as well.

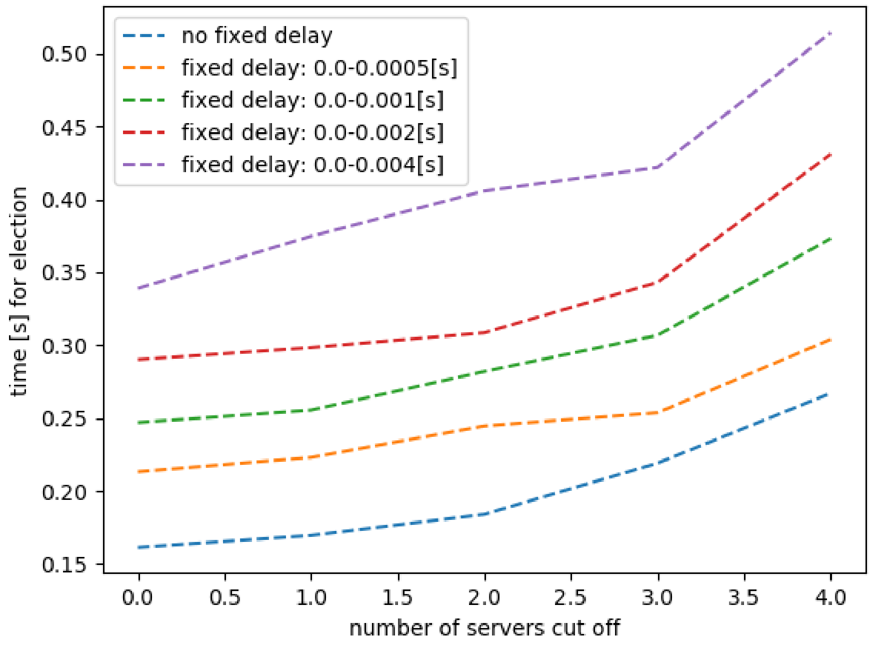

Figure 3.9: Comparison between the different fixed delays and partitions (10 servers, timeouts: 0.1-0.15[s], 'fake' partition)

In Figure 3.9 wanted to analyse the same as before, but with fixed delays instead of the fixed failures. We observe that fixed delays worsen the performance the more servers are cut off and the higher the fixed delays get but not that drastically as with fixed failures. We explain the increase of the curves with the fact that higher fixed delays make it less likely that a server reaches all the other servers before another server timeouts as well.
I.e. if a server sends a message to another server with a delay of 0.1[s] and an actual transmission time of 0.05[s], then no other server should timeout in 0.15[s] instead of 0.05[s]. Therefore, the larger the fixed delays get, the larger the

timeout interval should be to reduce the probability of the interfering Candidates again.

3.5 Discussion

The sections 2.2 and 3.3.1 showed that it is quite complicated to get a reliable simulation for a larger number of servers on one machine. This fact limited the whole range of interpretations but still we could find some answers to the two main issues:

- Isolated server: The original RAFT algorithm revealed some severe problems in such configurations. An isolated Follower continuously slows the whole process down. When the isolated server is the Leader, the situation is even worse. The Leader will not make any progress and does not give up its leadership which ends up in sort of a deadlock. In an open network without secured IP addresses one could put a malicious router near one of these servers. The situation of an isolated server now appears when this router discards all of the messages to this server close by.
  However, the problem could be completely handled by these policies AppendEntryReply Timeout, Commit Timeout and LastLeaderTermPolicy. The Commit Timeout prevents that a isolated server remains in the Leader state by giving its leadership voluntarily up after some time without progress. The LastLeaderTermPolicy makes sure that an isolated Candidate does not interrupt the progress of the others anymore by adding an additional VoteRequest restriction.

- Partition: Even if the policies IncreaseTimeout and IncreaseCandidate-Timeout seem to improve the progress quite a bit, it remains the most involved issue. It is still not very clear how well the algorithm works with a larger number of servers as the simulation does not really give a reliable answer to that.
  One obvious solution is to directly increase the upper timeout boundary linearly as the number of servers increase. In this way the timeout density stays the same and we assume that it remains likely to find a leader, even for a large number of servers. The election time will not increase either as the change of having an early timeout remains roughly the same. However, there will be servers with huge timeouts and this will make the algorithm slower and less adaptable to failures. I.e. a link failure may not be recognized until the time of the upper timeout boundary elapsed.
  Additionally, with the same probability of sudden server and link failures there will happen linearly many more elections which consumes time and in the worst case could end up in a constant leader election without any progress. As well it can happen that a Leader only has access to half of

the Followers and still remains Leader forever. As a consequence it is questionable if the guarantee of having only half of the servers up to date is effective enough.

After these observations it seems to us that RAFT has still some problems in some edge cases and it remains the question of how well the algorithm scales. However, it looks like a good theoretical approach which also leaves some freedom for further practical improvements, as we have already proposed some.

One interesting proposal is left for further analysis. Similar to the principle of 'divide and conquer' the RAFT process could be divided into different hierarchical sub-processes where the Leader of each sub-process represents a server node for the next upper process. To enable multiple levels, the servers have to be put in different predefined groups of different levels.
I.e. for 125 servers there could be 3 layers. The layer 1 group consists of all servers. The layer 2 group of 25 servers each and the layer 3 group of 5 servers each. Then every server belongs to a group in each layer (i.e. server 6 could then be in the first group of layer 1 and 2 and in the second group of layer 3). The servers in each layer 3 group will try to find a Leader. Then the Leaders of layer 3 try to find a Leader in their group in layer 2 and so on.

In this way RAFT potentially becomes more resilient to sudden failures as only one sub-process has to deal with it while the other processes will not be affected at all. The cluster membership changes, which are explained in the paper [3], would not need to be done globally anymore but only in the specific sub-processes. Additionally, the work of one global Leader of distributing the commands to every single server could be assigned by many other servers.

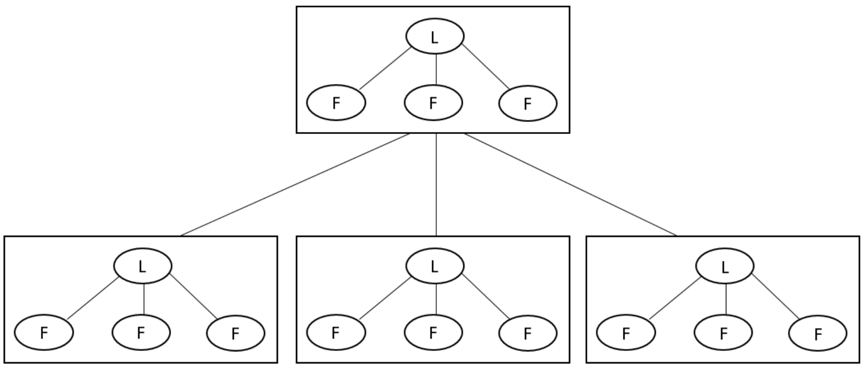

Figure 3.10: One idea of multi-RAFT. Each rectangle contains servers of a group of layer 2 with the according elected Leaders. The Leaders of layer 2 then find a Leader for layer 1 which then will be Leader for two layers.

Bibliography

[1] Lamport, L.: The part-time parliament. (1998)

[2] Lamport, L.: Paxos made simple. (2001)

[3] Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm, USENIX Association (2014)

Appendix A

Appendix Chapter

A.1 Sliding Window between client and server

The sliding window between client and server we implemented as follows:
On one hand there is a sliding window for all the commands which the client wants to append to the Leader. Every so called *AppendCommand* gets buffered on the client side and the client stops sending new commands as soon as the size of this buffer AppendBuffer is equal to the maximal size of the sliding window. When commands are left over to be sent to the Leader, they will be buffered in another buffer *WaitingBuffer* which stores all waiting commands. As soon as there is some more space in AppendCommand, the next command from WaitingBuffer will be sent to the Leader and transferred to the AppendBuffer.
On the Leader side there is a buffer called *WindowClient* which has the size of the maximum size of the sliding window minus one. This minus one comes from the fact that the first element (the one that actually is expected from the Leader) of the sliding window does not have to be stored as it can be appended to the log directly. So, whenever there is another than this expected element received, it will be stored at the right place in this buffer. As the expected element is received, the buffer has to be updated where every consecutive element after the received one can be applied as well. The other commands in WindowClient have to be moved that according amount of places forward in the buffer.
The Leader will send a feedback to the client which tells it the last applied command. This is also the case if the Leader receives a stale command which was already applied. Additionally, it sends a feedback whenever a new command got committed. The client on the other side can use this feedback to update AppendBuffer. The elements which are removed from this buffer are stored in another third buffer, named *CommitBuffer*. This is necessary as the Leader could crash and the appended commands might not be committed. In this case the commands would be lost. The client can use the second feedback about the committed commands from the Leader to remove the committed commands from CommitBuffer. Then it can be sure that it does not have to send these commands again.

Between client and Leader there could also drop some messages or a Leader could fail. In such cases it is important that the client sends its commands again. Therefore we introduce two timeouts. The first timeout *LastAppend* will go off whenever a certain time elapsed since the last positive AppendCommand reply from a Leader. When this timeout is issued, it forces the client to send all its AppendCommands again. There is a second timeout *LastCommit* which checks the last commit feedback from the Leader. When this timeout goes off, it will force the client to put all commands from CommitBuffer and AppendBuffer back to WaitingBuffer (preserving the order). The client then starts to resend commands and to refill the AppendBuffer again. These two timeouts make sure that RAFT does not get stuck between client and Leader. Without the second timeout it could happen that the client sends all its commands to an isolated Leader and this one never commits anything. Additionally, the commands will be sent to randomly chosen servers whenever the LastCommit timeout goes off or there is no Leader known.
To accelerate the whole sending process, the Leader sends a special feedback whenever it gets a message which needs to be buffered. It indicates that the element, which it needs as next, got lost. The client will then send this element again.