Improving Raft When There Are Failures

Short Paper

Christian Fluri
Distributed Computing Group
ETH Zurich
Zurich, Switzerland
fluric@student.ethz.ch

Darya Melnyk
Distributed Computing Group
ETH Zurich
Zurich, Switzerland
dmelnyk@ethz.ch

Roger Wattenhofer
Distributed Computing Group
ETH Zurich
Zurich, Switzerland
wattenhofer@ethz.ch

***Abstract*—This paper investigates the Raft consensus algorithm in the presence of failures. We are especially interested in how the single failures - link failures, isolation and partition - affect the running time of the leader election, which is an important building block of Raft. Our tests show that such failures are non-negligible. We therefore propose new timeout policies which can improve the performance of Raft.**

***Index Terms*—fault tolerance, Raft, network partition, consensus**

I. Introduction

When introducing the CAP theorem [2], Eric Brewer convincingly argued that real world systems should also respect availability. Since then, the distributed systems community is not only focusing on consistency, but also the other two letters in "CAP": partition tolerance and availability.

Various newer protocols have emerged, some allowing byzantine failures, others allowing more benign failures such as crash failures. Systems that only allow crash failures provide weaker correctness guarantees, but in reality they show better performance. The biggest impact on the crash-failure side was made by the Raft protocol [11]. By now hundreds of implementations of Raft exist.

In this paper we ask how efficient Raft really is in light of different failure scenarios. In particular we look at (i) link failures, (ii) isolated servers, and (iii) full network partitions. We introduce a novel extension of Raft: ReplicaRV, which we compare the original Raft simulation.

II. Background

The first state replication algorithm that could deal with server and link failures was Paxos [7]. This algorithm did however not get much attention until a new and simplified version of it was presented in [8]. Since then, Paxos was extended to different versions of Multi-Paxos [5], [9], [12] which all dealt with more clients. However, the Paxos protocol has been argued to be difficult to understand and the algorithm therefore exists today in many simplified writeups, e.g. [10]. In 2013, the Raft consensus algorithm was introduced in [11] where the authors claimed to have significantly simplified the state replication for multiple requests and multiple servers. It can be argued that Raft is just another implementation of Multi-Paxos. Instead of leaving the timing issue as an exercise to the user, Raft describes the timing of messages in great detail, and as such is directly implementable. Several similar approaches exist [1], [4], [6]; PiChain [3] for instance has a similar timing behavior as Raft, with the additional advantage that it does not need any explicit leader election, and as such may be robust during server isolations and partitions. In this paper we concentrate on Raft which seems to be the most popular implementation as of today.

A. Short Introduction to Raft

The Raft consensus algorithm simplifies the well known Paxos algorithm by introducing leaders who are allowed to dictate the state of the system in contrast to the the two-phase commit of Paxos. In the following, we only present the five main ideas of the protocol that are needed for further understanding of this paper. For more details the interested reader is referred to the original paper [11].

1) Leader Election: A server can be in one of the three states: *follower*, *candidate* and *leader*. At the very beginning, each server starts as a follower. It waits until it either receives a heartbeat message from a leader or its local clock reaches a *timeout*. A server that has reached a timeout tries to become the leader and therefore transforms to a candidate state. A candidate initiates a leader election by asking each server for its vote. All servers must give their vote to the first valid candidate that 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. In order 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. Potential candidates will return to the follower state once they receive a heartbeat from a current leader.

2) Terms: The time of the algorithm is divided into terms and it is a local integer value stored at the server. In order to initiate a leader election, a candidate will increment its term number and claim to be the leader of this new term. As the whole environment is asynchronous, two servers do not necessarily have an equal term number at the same global time. Whenever a server receives a message with a newer term, it will update its own term to the newer one and become a

978-1-5386-8489-4/18/$31.00 ©2018 IEEE

follower. This way, there will be only one possible leader in a given term. We use terms as a measure of efficiency in this paper.

3) 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. 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 committed and the leader can apply the command to its state machine. A committed command will eventually be executed on all servers.

At all times a follower has to check 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, then stepwise ask for the missing messages and append them. By 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.

4) Consistency: In order to maintain consistency of the committed commands among the servers and clients, a server which does not contain all the committed log entries has to be prevented from winning an election. The followers therefore only vote for candidates that are consistent with all their committed log entries. Since the committed entries are already present in the majority of the servers, only candidates with all committed log entries have a chance to win an election. 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.

5) Messages: For the Raft implementation only four message types are required:

- RequestVote: A message sent by a candidate to initiate an election and to ask every server for its vote. The reply, which contains a positive or negative vote, is denoted by RequestVoteReply.
- AppendEntry: A message responsible for the log replication between the leader and its followers. Whenever a log entry has to be added at the follower, the leader sends an AppendEntry message which contains one log entry. Then the follower will send an AppendEntriesReply message to the leader to verify that it appended the entry. If there is no more log entry to be added, the leader sends an empty AppendEntry message which serves as a heartbeat.

B. Our Contributions

The running time of the Raft protocol depends on the number of terms needed for a successful leader election. Link failures, isolation of a single server and network partition can however influence the number of rounds needed to select a leader as we will explain next.

1) Link Failures: In practice, links between servers can have a non-negligible failure rate. In such an environment many leader elections fail due to the lack of received RequestsVotes and RequestVoteReplies and the chance for a candidate to win a leader election decreases. The original algorithm has no mechanism to effectively react to this issue and thereby the whole progress is significantly slowed down with higher failure rates. As there are two successfully received messages needed for one vote – RequestVote and RequestVoteReply – we expect a non-linear increase of the required terms for a leader election for increasing failure rates.

2) Isolation: A server that gets isolated from the rest of the system might influence the other servers in different ways. Assume that the isolated server is the leader of the current term and its messages can reach the remaining nodes. Such a leader can append new entries to the other server's log but it will not know if its followers could successfully append them. Without feedback, the leader cannot commit any new commands but also the followers will keep receiving heartbeats and will thus not initiate a new election, which results in a deadlock. An isolated server that is in a candidate or follower state can also interrupt the progress. As the isolated server does not receive any AppendEntry messages from the others, it will repeatedly timeout, increase its term and start a new leader election. Therefore, it might interrupt the progress of the system by forcing other servers to update their term and restart as followers. Consequently, other servers have to find a new leader and commit new log entries before the isolated server initiates the next leader election.

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. In this case the partitions will run two separate leader elections simultaneously. Note however that only the partition with the majority of the servers can be successful.

In this paper we analyze how much the failures affect the running time of the protocol and suggest new ways in which additional and more adaptive timeouts can improve the running time of Raft.

III. Implementation

In order to keep the implementation simple, we implemented Raft following the instructions in [11]. We run the simulation on one machine. Therefore, we created all server processes as independent threads and let them communicate via sockets.

As programming language we have chosen Python 3.6, since it provides a threading library with a fair distributed scheduling in terms of CPU allocation. The library used for the asynchronous messaging is ZeroMQ.

For our implementation we needed to guarantee that the messages are sent and received in a fair way and in as little time as possible. In order to also achieve scalability, we decided to place a socket listener in front of each socket. For each socket listener there 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 server can then listen to the messages in the queue in a FIFO (First In First Out) manner.

Threads are essential to provide concurrency in a distributed system. For every server we therefore use one main thread which reads the message queue and reacts according to the messages, i.e. it sends a reply to the leader. Additionally, there is one thread for each socket and a thread for each timeout that is needed to warn a server in case of a timeout. This works analogously for the clients.

For the simulation of link failures a stochastic bidirectional model is used, e.g a link failure of 0.1 means that every message on that specific link is lost with a probability of 10%.

IV. Evaluation

In this section we will address the three issues of the Raft protocol that were pointed out in Section II-B and discuss how they may be resolved.

1) Link Failures: We analyze the algorithm with non-negligible failures between the servers as described in Section II-B1. Therefore we compare the two policies, Replica RequestVote and Replica AppendEntry, with the original implementation.

We built a naive policy called Replica RequestVote, or short ReplicaRV, which dynamically deals with failures. The idea is to send the RequestVote and the corresponding reply messages several times. The number of times a message is sent is equal to the number of terms that passed since the last known leader was active, e.g. if the candidate starts an election in term 4 and its last known leader was active in term 1, it sends each message 3 times.

With this policy some elections may still fail, but with increasing number of terms the messages are sent so frequently that at least one of the messages is expected to reach its recipient.

Figure 1 shows that without ReplicaRV the number of required terms increases non-linearly with respect to the number of general link failures. With ReplicaRV the number of terms remains considerably lower. We can conclude that even a quite naive mechanism like ReplicaRV significantly improves the leader election.

2) Isolated server: We first consider the case where the isolated server starts as the leader. In order to resolve a deadlock in this case we add the following new mechanism:

- Commit Timeout: We add an additional timer to the leader which will timeout whenever no more log entries (if existing) have been committed within a certain time interval. This kind of timeout indicates that the leader cannot access a majority of the servers anymore and this server is forced to give up its leadership and restart as a candidate.

In fact, this policy works for any isolated cluster, consisting of a minority of servers, including a leader. Due to the lack of progress, the isolated leader will always be reset to the 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. Therefore, the Commit Timeout does not interfere with normal configurations.

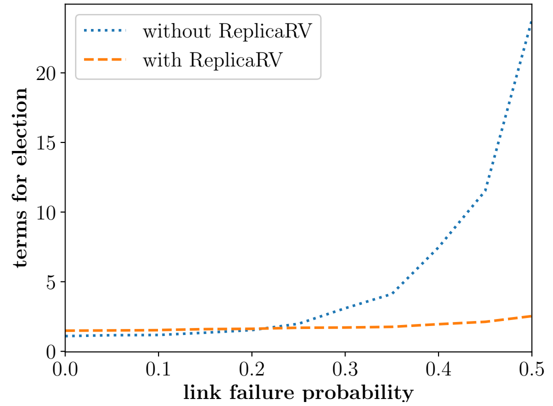

Fig. 1. The impact of the Replica RequestVote on an election with non-negligible link failures (timeouts: 0.1-0.15[s], servers: 10, 200 trials)

The second case consists of an isolated candidate. One solution to prevent interruptions by such a server is the so called LastLeaderTerm Policy. To the already existing safety requirements in the leader election, which guarantee the last log entry of the candidate to be at least 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 will not update its term. If the LastLeaderTerm of the candidate is higher or equal, the follower proceeds with the RequestVote as normal. However, this policy violates the safety guarantees of Raft and cases can be created that end up in a dead-lock with just one server failure. Consequently, the LastLeaderTerm Policy should only be applied to followers that currently receive heartbeats from a viable leader, i.e. the LastLeaderTerm is equal the current term. With this restriction it can be shown that the safety requirements are still met.

3) Partition: We consider the network partition problem for a small number of 10 servers. For a large number of servers, the simulation effort increases such that the leader election cannot be finished successfully anymore. Therefore, instead of increasing the number of servers, we can try to make the lower and upper timeout bounds tighter, e.g. if we halve the timeout interval, the density of timeouts will be doubled. This simulates a situation where the timeout interval remains and the number of servers gets doubled.

In Figure 2 we simulate 10 servers with a varying number of servers that are cut off and vary the upper bounds on the timeouts. The results show that the election time continuously

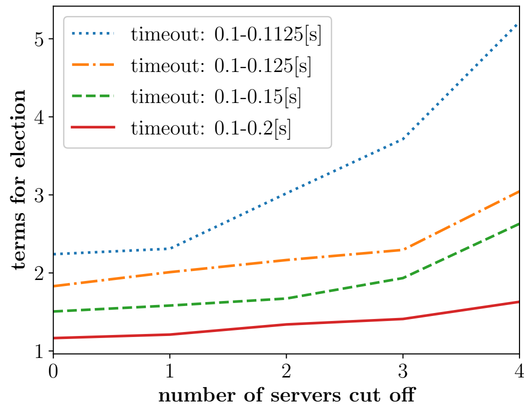

Fig. 2. Comparison between different timeout intervals (10 servers, 200 trials)

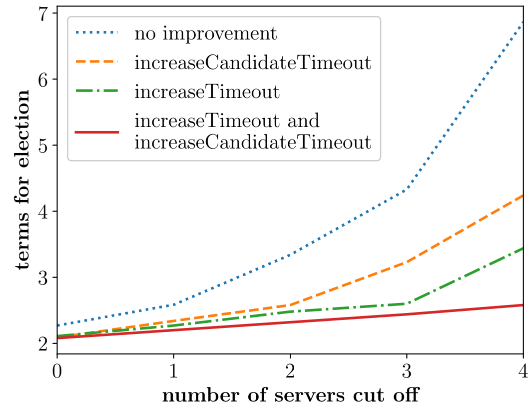

Fig. 3. Comparison between the different timeout policies (timeouts: 0.1-0.10625[s], 10 servers, 200 trials)

increases as the timeout interval gets halved. For 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 by the increasing timeout density, when the timeout bounds get tighter.

The number of servers to be simulated seems to be limited for a fixed timeout interval. When the number of servers is 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 large number of servers. To deal with this issue, we introduce two possible approaches:

- increaseTimeout: Each server remembers the last leader term and counts the consecutive terms without a leader. Every time there is a new term without a leader, it increases its upper bound of the timeout by the original interval length of the timeout (upper bound minus lower bound). This way the interval increases linearly until a new leader is found. This can be done as well with an exponential increase of the boundary, but the method proved to perform worse in our experiments.
- increaseCandidateTimeout: The candidate counts the number of positive and negative votes it has received. Then it adapts its upper timeout bound for the next timeout according to the ratio between the positive and negative replies. Thus, 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. This strategy gives an advantage to the more promising candidates.

Figure 3 compares the performance of Raft with the introduced policies. We used 10 servers and a tight timeout boundary of 0.1-0.10625[s] where we varied the partition size. The graph shows 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. The two policies together seem to improve the progress even more, which indicates that the policies are not making the same improvements.

A. *Discussion*

The original Raft algorithm revealed some severe problems in configurations with link failures and isolated servers. These problems could be resolved efficiently by adjusting the timeout lengths and adding new timeouts for leaders, enabling a leader to resign from its current position. The partition remains the most involved issue. It is still not clear how well the algorithm works with a larger number of servers as our simulation does not provide a reliable answer.

One possible solution is to directly increase the upper timeout boundary linearly as the number of servers increase. 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 large timeouts that would slow down the algorithm and make it less prone to failures.

References

[1] Mahesh et al. Balakrishnan. CORFU: A Shared Log Design for Flash Clusters. NSDI'12.

[2] Eric Brewer. CAP Twelve Years Later: How the "Rules" Have Changed. *Computer*, 45(2), February 2012.

[3] Conrad Burchert and Roger Wattenhofer. piChain: When a Blockchain meets Paxos. OPODIS 2017, pages 2:1–2:13.

[4] Mike Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. OSDI '06.

[5] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. PODC '07.

[6] James C. et. al Corbett. Spanner: Google's Globally-Distributed Database. OSDI '12, pages 261–264.

[7] Leslie Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998.

[8] Leslie Lamport. Paxos made simple. 2001.

[9] David Mazieres. Paxos Made Practical. 2009.

[10] Meling, Hein and Jehl, Leander. Tutorial Summary: Paxos Explained from Scratch. Springer International Publishing, 2013.

[11] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. USENIX Association, 2014.

[12] Robbert Van Renesse and Deniz Altinbuken. Paxos Made Moderately Complex. *ACM Comput. Surv.*, 47(3), 2015.