Byzantine Fault Tolerant Raft

Dennis Wang, Nina Tai, Yicheng An
{dwang22, ninatai, yicheng} @stanford.edu
https://github.com/g60726/zatt

Abstract

For this project, we modified the original Raft design to be able to tolerate Byzantine faults. In this paper, we give a brief overview of Raft and describe our implementation and modifications, which are influenced by the design of PBFT. In the end, we give an evaluation of safety and liveness for our modified Raft protocol.

1. Introduction

Raft[1] is a consensus algorithm that uses log replication. In an effort to make a more understandable consensus algorithm, it separates the key elements of consensus into leader election, log replication, and safety. While Raft is a relatively easy to understand and implementable consensus protocol, its safety guarantees only apply to fail-stop behavior for servers. In practice, even a single adversary taking control of a single server would be able to make the protocol unsafe. Hence, we aim to enhance the original Raft algorithm such that it becomes tolerant to these types of Byzantine faults[1].

To do this, we apply techniques from Practical Byzantine Fault Tolerance[2] (PBFT). Our goal is to make Raft Byzantine fault tolerant while still maintaining its simplicity and understandability as a protocol. We start by presenting an overview of Raft and PBFT in section 2. Next we describe our implementation of the leader election and log replication protocols in section 3. For leader election, we modify Raft to use round-robin candidate selection and to require a quorum to start an election, where quorum size is 2f+1 when the total number of servers is 3f+1. For log replication, we added in a prepare and commit phase between the leader and followers. We have also updated all steps of the protocol to use cryptographic signatures for authenticated communication and proof. In section 4, we revisit the safety and liveness properties provided by Raft in terms of Byzantine failures. In sections 5 and 6, we conclude with a critique of our design in terms of future work and a summary.

2. Related Works

2.1 Raft

Raft is a consensus algorithm, which means it allows a collection of machines to work as a coherent group that can survive the failures of a few of its members. Compared to Paxos[3], Raft has a stronger form of leadership in the sense that client only talks to the leader and only the leader decides what to add to the log.

[1] Note that we do assume the client is non-faulty and the adversary cannot take control of network.

Raft uses randomized timers to elect leaders for each term. Once a follower times out from not receiving leader heartbeats, it becomes a candidate for the next term. It will send out requests to all other servers and become a leader if a majority of servers agree. If no leader is elected in a term, candidates will time out and start another election for the next term. In order to preserve safety, only a server with the most up-to-date logs can become a leader. This guarantees the Leader Completeness Property, which states that if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.

During normal operation, the leader is responsible for appending log entries based on client requests. Once the leader receives a request from client, it will append the command to its log as a new entry and issue *AppendEntries* RPCs to follower servers. Once a majority of servers receive and acknowledge the new log entry, the leader applies the entry to its state machine and returns the result of that execution to the client. The leader also maintains a *nextIndex* for each follower, which is the index of the next log entry the leader will send to that follower. If a follower's log is inconsistent with the leader's, the follower will reject the *AppendEntries* RPC, and the leader will decrement *nextIndex* and retry the *AppendEntries* RPC. This allows the leader to remove any conflicting entries in the follower's log and bring it up to date.

2.2 Practical Byzantine Fault Tolerance

Our modifications to Raft draw inspiration mainly from PBFT. Like PBFT, we describe an algorithm that offers both liveness and safety provided at most f Byzantine failures out of a total of $3f+1$ servers.

In PBFT, replicas move through a succession of configurations called views. In a view, one replica is the primary, and the rest are backups. The primary of a view is the replica with id $p = v \bmod R$, where v is the view number and R is the number of replicas. In order to start a view change, there must be at least $2f+1$ replicas requesting it. With this design, PBFT ensures liveness by preventing faulty replicas from forcing frequent view changes and from always promoting a faulty node to primary. We apply the same design choices to our Raft extension, except with election terms instead of views and leader/followers instead of primaries/backups. To provide safety, PBFT uses a three-phase protocol: pre-prepare, prepare, and commit. The pre-prepare and prepare phases are used to totally order requests sent in the same view even when the primary is faulty. The prepare and commit phases are used to ensure that committed requests are totally ordered across views. We make a similar modification to the Raft protocol, except we maintain Raft's simplicity by coordinating these phases through the leader instead of through a series of mass broadcasts.

3. Implementation

Our implementation is based on a publicly available Raft implementation that is written in Python, called Zatt. Zatt provides a key-value store on top of a cluster of servers coordinated by

Raft that can be simultaneously accessed by multiple clients. It turns out that there were numerous bugs in the original implementation, so we got bonus experience in implementing and debugging basic Raft. In this section, we will describe our implementation and the changes we made to each part of Raft to allow it to withstand *f* Byzantine faults in a cluster of *3f+1* nodes.

3.1 Leader Election

As described above, it is imperative that we ensure liveness during the leader election process. In our implementation, leader election occurs in the following steps (where R=3f+1):

Step 1: A server times out and sends a *startElection* request to server id $n = t \bmod R$, where *t* is the new election term.
Step 2: A server receives *2f+1 startElection* requests and becomes the candidate for the election for term *t*. It broadcasts a *requestVote* message containing *t*, the *2f+1* signed *startElection* requests for *t*, and its most recently prepared log entry.
Step 3: A server receives the *requestVote*, ignoring it if the term is lower. If the term is higher, it verifies the *2f+1* signed *startElection* requests and converts to a follower. It verifies the attached log entry and checks to see that it is at least as up-to-date as its own prepared log entries. If so, it replies with a *grantVote*.
Step 4: The candidate receives *2f+1 grantVote* messages. The candidate becomes the leader and immediately sends *appendEntry* to each other server with the *2f+1 grantVote* messages as proof that it is leader.

For step 1, term *t* is incremented after each timeout. A timeout can occur because a follower does not receive an *appendEntry* from the leader, because a candidate is unable to get the requisite number of *grantVote* messages, or because a client broadcasts that a request has timed out. There is only one valid candidate per term. By using the equation candidate $n = t \bmod R$, we effectively round-robin through the nodes in the cluster as candidates. For step 2, a server waits for a quorum of *startElection* requests in order to ensure that the Byzantine servers are not able to trigger continuous election cycles. For step 3, just like normal Raft, a server may refuse to grant its vote if the candidate does not have the most up-to-date log. Also, by ignoring lower term vote requests and deferring to higher term vote requests, we effectively allow the highest valid election to always win. For step 4, once a server hears *appendEntry* with valid proof and higher term, it immediately converts to a follower and updates its term accordingly.

3.2 Log Replication

We modified the log replication process of Raft as follows to avoid faulty log append operations from a potentially fault leader.

Step 1: The client issues a request to the leader of the current term.
Step 2: The leader broadcasts *appendEntry* to all followers with a copy of the signed client request and the log index to append the entry to.
Step 3: Follower receives *appendEntry* from leader and replies with *appendEntryAck* if terms match, the client request is properly signed, and there is not already an entry prepared for the proposed log index.
Step 4: The leader collects *2f+1 appendEntryAck*'s. It persists these messages and broadcasts them to all followers in a *prepareEntry* request.
Step 5: Follower receives *prepareEntry* from leader and replies with *prepareEntryAck* if terms match, there is a sufficient amount of signed *appendEntryAck*'s as proof, and there is not already an entry prepared for the proposed log index. Before replying, it persists the *appendEntryAck*'s it has received.
Step 6: The leader collects *2f+1 prepareEntryAck*'s. It persists these messages and broadcasts them to all followers in a *commitEntry* request and replies to the client with *reqFinished*.
Step 7: Follower receives *commitEntry* from leader, verifies the *prepareEntryAck*'s as proof, persists the *prepareEntryAck*'s, commits the entry to the log, applies it to the state machine, and replies to the client with *reqFinished*.

In step 1, if the server is no longer the leader, it will forward the request to the current leader. The client application will try to contact other servers randomly, if there is still no response. Step 2 is essentially the pre-prepare phase in PBFT. This phase helps prove that the leader is issuing the same request to all servers for the same log index. Step 4 is essentially the prepare phase in PBFT. It ensures that all followers persist the proof that an entry is already prepared for the given log index. Step 6 is the commit phase of PBFT. It allows the followers to externalize the results of the entry and reply to the client. Steps 3 and 5 are checks made by followers to ensure the integrity of the leader's message and to ensure that a previously prepared entry is not overwritten.

3.3 Cryptographic Signatures

During cluster setup, we pre-configured each server and client with its own unique private key as well as the set of public keys for all other servers and clients. We used elliptic-curve cryptography for asymmetric signature and verification. Whenever a message is sent in the network, whether by server or client, it is signed with the sender's private key. By using the sender's public key, any receiver is able to verify the identity of the sender of the message it receives. This prevents Byzantine servers from masquerading as other servers or as clients.

We also leverage these signatures as proof of the leader properly completing each step in the log replication process. For example, a copy of the client's signed message is included in step 2 to prevent the leader from forging fake requests. In steps 4 and 6, the *2f+1* signed ack's are persisted as proof of prepare and commit, respectively. This is essential to prove to servers

during log replication that a particular entry is safely committed to a certain log index. In the future, any servers that receive log updates from the leader need only to verify the proof sent along with the log update to be able to safely apply the update.

3.4 Client Application

For the client application, we implemented a local TCP server that is able to perform retransmits and timeouts for requests to the Raft cluster. The user only needs to connect to the application, and it will proxy any requests (in this case, key-value store) to the Raft cluster. One notable difference in our client implementation with the basic Raft implementation is its ability to verify *f+1* matching responses for a given request id before confirming the success of the request. This provides resilience to *f* Byzantine servers. Also, upon request timeout and exceeding of maximum retransmits, the client will broadcast the timeout to all servers in the Raft cluster, thereby triggering a leader election.

4. Safety and Liveness Properties

4.1 Safety

As in Raft, we guarantee that if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. We do this by leaving the most up-to-date log requirement in the leader election process intact. This is referred to as the Leader Completeness Property of Raft.

To account for a Byzantine leader, we adapt the three phase protocol used by PBFT. Just like PBFT, all messages are signed. The first phase (pre-prepare) ensure that the leader issues the same request (entry + log index) to a quorum of servers and that the request is not forged. The second phase (prepare) begins with verification of *2f+1* acknowledgements from phase 1 to ensure that there is an agreement on the entry and location to append. The acknowledgements from this phase ensure that the followers have persisted this proof. The third phase (commit) begins with a verification of *2f+1* acknowledgements from phase 2 to ensure that a quorum of followers have persisted the proof of the entry and location. This means that these followers will reject any future pre-prepare and prepare messages for this log location, thereby preventing another quorum from overwriting the entry. At this point, the servers are notified that they are safe to commit and externalize the results of the log entry.

To account for Byzantine servers, the client waits for *f+1* reponses for any read or write operation. This ensure that at least one good node was told that it was safe to externalize the results.

4.2 Liveness

When doing leader election, we enforce two rules. First, there will only be one candidate per term, and the nodes are round-robined as candidates for the terms. Second, to start an election, a server must collect *2f+1* election requests. The first rule provides the same liveness

guarantee as PBFT, where the leader cannot be faulty for more than *f* consecutive terms. The second rule ensures that leader elections cannot be started arbitrarily by Byzantine nodes. To ensure that a faulty leader does not retain control of the system indefinitely, the client will broadcast a leader election request to all of the servers if its requests are not being serviced properly.

4.3 Testing

In order to test the Byzantine resilience of our implementation, we implemented a chaos server that periodically does the following. It randomly selects a message type (i.e. *appendEntryAck*, *grantVote*, etc), forms it with the proper fields, gives it an up-to-date term number, stuffs random data into any remaining fields, and properly signs it with the node's private key. It then broadcasts this message to the entire Raft cluster, in the hopes of interfering or crashing existing message exchanges and protocols. Our tests show that our implementation is resilient to at least this baseline of failure.

5. Future Work

Our implementation is complete in the sense that it satisfies our design for the safety and liveness properties of Raft and PBFT. Our focus was on correctly implementing these properties and not on optimizations. As such, there are still a lot of optimization that can be done. In Raft, it is mentioned that the number of rejected *AppendEntries* RPC can be reduced in various ways, such as doing a binary search. Other optimizations like log compaction to save space would also be useful. Also, with our three-phase log replication protocol, it is in principle possible to "prepare" multiple entries at once. However, we simply did not have the time to look into and demonstrate this.

6. Conclusion

In summary, we chose Raft as the baseline for our Byzantine fault tolerant system because of its relative ease to understand and implement. Raft neatly divides the consensus problem into leader election, log replication, and safety. Likewise, we approached each of these aspects independently when constructing our extension, drawing heavily from the principles of PBFT. In the end, we were able to create a demo distributed key-value system that could tolerate continuous interference from our Byzantine chaos server.

7. References

[1] Ongaro, D. and Ousterhout, J. (2017). *In Search of an Understandable Consensus Algorithm*. Stanford University.

[2] Castro, M. and Liskov, B. (1999). *Practical Byzantine Fault Tolerance*. Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139.

[3] Lamport, L. (2001). *Paxos Made Simple*. [Book].