A Byzantine Fault Tolerant Raft

John Clow
Stanford University
jclow@stanford.edu

Zhilin Jiang
Stanford University
zjiang23@stanford.edu

Abstract

Just like how Paxos is hard to understand, PBFT is also hard to understand. Therefore, we want to formulate a Byzantine Fault Tolerant distributed file system that is easy to understand. Therefore, we took inspiration from Raft, PBFT, as well as Blockchains (because it's the latest fad) to create a simple BFT algorithm.

1. Introduction

In the world of high stakes distributed computing environments, fail stop semantics is a bad assumption to rely on. Even though oftentimes servers in the distributed system are not expected to exhibit malicious behavior, it is possible that faults cause them to have arbitrary behavior, and therefore a distributed system that assumes fail-stop semantics is vulnerable to bugs, hardware faults, and other problems that can be exhibited by either the servers or the network. Therefore, in order to give higher guarantees on correctness even in the case of these failures, a Byzantine Fault Tolerant (BFT) distributed computing environment is desirable.

Additionally, the more complicated an algorithm is, the more places where subtle bugs could occur. Therefore, in order to have the highest guarantees on correctness, one should strive to find the simplest algorithm that achieves all of the targets, even if that algorithm requires some trade offs. The current PBFT algorithm [1] has some pain points, such as the method for catching up the state when there is a view change, and the log index reservation process. Therefore, we want to introduce a system with slightly less performance optimizations, that will in turn be easier to understand and easier to implement correctly.

2. BFT Raft Algorithm

In order to make the Raft consensus protocol [4] implementation Byzantine fault tolerant, there must be many significant changes to the algorithm.

2.1. Signatures for Verification

Without any protections, Byzantine fault tolerant nodes can easily masquerade as other nodes. Therefore, in order to prevent spoofing and replays, all communication from a node must be signed.

In our implementation, we will be using digital signatures for node verification. As the cryptographic landscape continues to change, new algorithms for client signing will emerge. We expect that our use of these signatures will be compatible with these future cryptographic primitives.

Current asymmetric cryptographic algorithms require that each server has its private key as well as the real public keys for all servers in order to securely create and validate messages. In order to transfer these signatures to all clients, we need either an out-of-band trusted transfer method to transfer the public keys, or a trusted server through which the keys can be communicated, just like PBFT.

Note that we assume that the client is correct. There is no way for a distributed system to distinguish between a legitimate client request and a malicious one, because every action can have either a proper intent or a malicious intent. For example, there can be legitimate and illegitimate cases for deleting a file, and we can't tell them apart beforehand.

2.2. Cryptographic Digests

In the Byzantine Fault Tolerance version of Raft, cryptographic hashes (digests) are used to generate a fingerprint for the actual data chunks. The use of hash-

ing allows nodes to verify their agreement on a certain value. In Raft-BFT, nodes transmit a hash value instead of the actual data chunk until the transaction is to be committed, thus drastically saving message overhead for aborted transactions. Hashing is also used to verify the consistency of the local states at two different nodes, thus ensuring that transactions are processed in the same order across them. Examples of hash functions that can be used for cryptographic digests include the SHA-2 family (e.g. SHA-256, SHA-512) and Keccak.

2.3. Leader Change

The RAFT leader election process is inherently not byzantine fault tolerant. A faulty node could ignore the timeout and trigger the leader election immediately, and through the coordination of two faulty nodes, they could switch leaders between the two nodes back and forth, preventing any work from being committed. Thereby our new leader selection algorithm is essentially the one from PBFT. There is a rigged leader election, that happens after a fixed timeout time per node, and each node votes in a way that increases the possibility of consensus forming.

Some differences between the PBFT algorithms are some details on the timeouts, and the first commit after the leader is selected. Timeouts in this RAFT-BFT happen when the leader has not made any forward progress (Appends or Commits) in a predetermined amount of time. Arguably this method is not ideal, a faulty leader can always slow down the algorithm by committing the least amount possible, however, the PBFT algorithm has its own faults, namely, under high load, the leader would timeout, causing frequent view changes which would prevent forward progress. Therefore, an ideal solution might be the combination of the two, which will be explored at a later time.

Unlike PBFT where there is a complicated algorithm to determine the entries that need to be rerun, in our RAFT-BFT, there are at most one entries that need to be rerun. Because of the constraints on our algorithm, we know that only one entries could be appended but not committed. Therefore we can just reapply the entry as normal after the leader change commit is approved.

2.4. Log Replication

Once a leader is selected, it and other clients can begin servicing client requests. For each request, a client sends a message to what it thinks is the leader with a command to be executed by the replicated machines, as it does in Raft, but it includes an additional client signature to ensure validity of the message. This prevents another node (including the leader) from spoofing the request.

After the leader receives the message, there are three phases, Pre-Append, Append, and Rollup Phases, as seen in Figure 1. The Pre-Append and Append phase is used to totally order the server commits and through that the client transactions, then the Append and Rollup phase are used to ensure clients and servers that the transactions have been properly appended to the logs.

If the client does not hear back about the request after a certain amount of time, it then broadcasts the message to all nodes. Any node that is not the leader and receives a message from a client forwards the message to what it believes is the current leader.

2.4.1 Client Leader Interaction

Before the Pre-Append phase, clients send to the leader transactions. Each transaction contains the transaction body, the timestamp of the transaction, a client identifier (an integer), as well as a signature from the client validating the message. The timestamp for the client is a monotonically increasing counter for the client.

The former allows for replay protection, as each new transaction needs to have a different counter value. The signature is used to validate that the client sent the message preventing any adversaries, including the leader from spoofing the client transaction.

When the leader receives the client message, it will add it to a queue of pending messages, and add it to stable storage. Periodically, the leader will group a set of the messages in the queue, validate all of them, and create a new Raft Transaction set.

The validation consists of:

- Check that the client signature is valid and matches the client number

Source	Dest.	Message
Client	Leader	$<$ NEW, $Transactions, t, c >$
Leader	Backups	$<$ PRE-APPEND, $l, n, d >$
Leader	Backups	(DATA, $[< Tx, t, c >_{\sigma_c}]$)
$Backup_i$	Leader	(PRE-APPEND-ACK, $i, sig_{pre-app}$)
Leader	Backups	$<$ APPEND-ENTRIES, $P(commit), d_c, P(a) >$
$Backup_i$	Leader	(APPEND-ACK, i, sig_{app})
$Replica_i$	Client	$<$ COMMITTED, $r, i, d >$

Table 1. A brief summary of the messages in the normal case commit. The triangle brackets ($<$ ... $>$) signify that the entire message is signed.

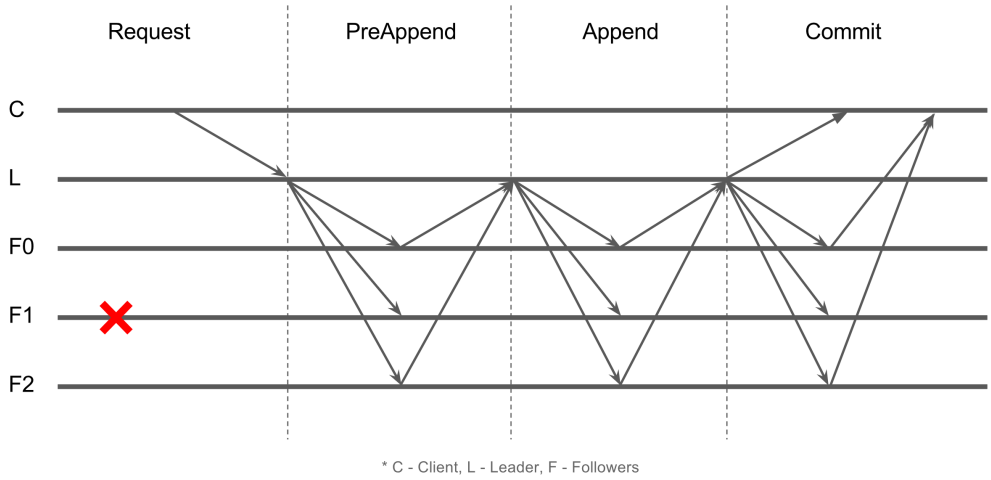

Figure 1. Normal case operation for Raft-BFT

- Check that the client timestamp is greater than all previous client timestamps
- Check that there are no conflicts between transactions
- Validate that the transaction is valid and well-formed

The leader then sends out both a Pre-Append packet (as detailed below), as well as the data packet. The data packet doesn't need to be signed, and on top of that does not need to be received by clients until the Append phase, allowing the leader to send the data with a lower priority.

2.5. Pre-Append Phase

After validation, the leader then combines all of the transactions into a single PRE-APPEND packet that includes the current leader term, the log entry number for this new entry, a hash of the list of of transactions in the entry, as well the leader's signature of the former entries proving that it created the entry. The leader signature ensures that no faulty node can create a fake but valid looking Pre-Append message that could stall the commit pipeline

When a backup receives the PRE-APPEND entry, it validates that the leader term and entry number combination is higher than any entry it has approved before. There is no need to validate that the entry number is only a bit higher than the previous entry, as that is done in the Append phase. . If the checks pass, the client adds the PRE-APPEND to its log. It then responds to the leader with the term and log numbers, the client number, as well as a signature signing the other elements in the packet along with the list of transaction hashes.

2.6. Append Entry

In order to make the protocol simple, there can be at most one entry in each of the Append Entry and

Rollup Phases. This does not hinder performance because multiple transactions are included in each entry that is being committed. This does reduce the amount of data transmitted as well. The Pre-Append stage has already ensured the data order which in turn means that there is no need for any entry numbers or other identifying data, which in turn saves a lot of data.

The leader creates one packet for each AppendEntries, which can include both information for the Append phase and for the commit phase. For the Append phase, the leader signs the Pre-Append data in the same way the backups do and after collecting a total of $2f + 1$ Pre-Append signatures, sends them as proof that the Pre-Append has succeeded. When there are no ready append messages, the leader can send the empty array for the signature field signifying so.

When the follower has received the Pre-Append signatures as well as all of the transaction packets for the next entries to append, and processed any commits in the last AppendEntries message, the follower first does the validation of the transactions in the same way that the leader does during the Pre-Append phase. Unless either the leader or the follower is faulty, this is guaranteed to succeed. Then, it validates that the hashes of the transactions in the Pre-Append is valid and that the follower signatures the leader attached as proof for the Pre-Append phase is also valid. It additionally validates that the entry index for the append is one higher than the one for the latest committed entry. After validating the entry, the follower then executes the command and then appends the entry into the log. Finally, after it has sent the response to the client for the previous entry (the one already committed) , it sends to the leader its node number a signature, signing the concatenation hash of the current Append entry along with the last entry back to the leader node.

2.6.1 Commit

In order to fully commit the entry, there's one more step. The leader again has to collect $2f + 1$ valid signatures, but these are append signatures. After doing so, it first stores it in a stable log, then replies to the client with the result, a hash of the client request, it's node number, as well as a signature signing the message. Finally it sends the signatures as part of the AppendEntries message to the backups. The backups upon receiving the message, also validate it, add it to their stable log, and send a reply back to the client. When the client has received $f + 1$ identical messages, it knows that the commit is successful.

3. Multiple Node Total Ordering Validation

The PBFT paper prevents a node from reordering the transactions of a client, but it doesn't protects against the leader reordering the requests from multiple clients relative to each other in order to cause unintended effects. For example, client A can request to move a file to a folder and client B requests to delete the entire folder, each after reading the contents of the folder. In PBFT, the primary could order these in arbitrary order, and thereby can arbitrarily decide whether the moved file is deleted.

We propose a simple extension to detect and prevent arbitrary results, that the client adds the leader era and the log entry number for the earliest transaction which the current transaction is dependent on (eg. the read of the directory), and both the leader and the backups validate that there are no conflicting actions between the earliest transaction and the current transaction. If a non-leader node detects the conflict, responds to the client and stops processing the request. If the leader node detects the conflict, it responds to the client, but continues to process the request, but using a special log entry value to mark it as a conflict and not prevent subsequent valid requests from going through. The client only accepts the conflict message when f + 1 nodes respond that there is a conflict. This ensures that at least one honest node validates the conflict, and prevents failed nodes from tricking the client into sending duplicate valid transactions and violating the at most once semantics.

4. Snapshots

In PBFT, the only ways to recover from a failure is to rely on the view change algorithm or wait for a snapshot and keep a log of all commits after the snapshot while the snapshot is being transferred. This is particularly problematic in intermittent networks, where one node could lose enough successive packets from other nodes that it would be unable to accept a commit, making it fall behind. In theory, if PBFT is implemented with signatures instead, a node could catch up without waiting for a view change or snapshot. However, even

in that case, the node needs to get the entire log since the last snapshot from at least $2f$ other nodes in order to guarantee validity. This is because f nodes may be faulty and can send the history with holes in it, and a further $f - 1$ other nodes might be out of date.

In our RAFT-BFT implementation, there are two easy ways to catch up: Retrieving past entries through tree-based logs, or using a Merkel tree to keep up to date snapshots. In the former method, like PBFT, snapshots are taken every n non leader change entries, n being a predetermined number (like 1000). Each of these entries are special entries, where the leader and the followers confirm that they have the same hash. Any entries before the snapshot can then be discarded.

The method for nodes to catch up is significantly simpler than PBFT though. Whenever a node hears of a committed entry hash from the leader's ApendEntries message that it doesn't recognize, it sends messages to its non-leader peers, asking for the associated with the hash. The peer responds with the entry, which includes all the data associated with the entry, the $2f + 1$ signatures proving the entry is real, and the hash for the previous entry. If the entry is a snapshot, the peer additionally responds with the entire contents of the snapshot. Through this, the node can verify the chain of commits, and can rebuild the history without relying on a new timestamp or view change and without talking to the leader. It doesn't usually need to talk to $2f$ nodes either, it only needs receive a single response with the previous commit.

4.1. Merkel Tree Snapshots

The other method of keeping snapshots is through using a Merkel tree to keep a constant hash of the state. This allows for basically a log free BFT implementation. In this formulation, the commit hash is replaced with the Merkel tree, which as before is afterwards secured through signatures. A node that receives a commit therefore knows the latest Merkel tree root, and can contact peers for the rest of the tree. In this case, the only state that the peers who are up to date have to store is the Merkel tree of the latest committed entry state, as well as all information associated with all non-committed entries and just the latest committed entry.

5. Implementation

We made two implementations, a inoperative one in Rust, and one in Python of just the typical case operation.

5.1. Rust

Our implementation is based on Raft-rs [2], an implementation of Raft using the Rust programming language. Raft-rs uses Cap'n Proto for its RPCs, and Mio (Metal I/O) for its asynchronous event loops. Raft-rs has existing unit tests and benchmarks available, which we had plan to leverage to ensure that the Raft API behaviors stay consistent after our modification. We began our implementation with the forked version by GitHub user paenko [5], which added several fixes on top of the original Hoverbear code base.

The Raft-rs code base, although marked to be only of alpha quality, provides a complete implementation of the Raft consensus protocol, including normal-case transaction handling, leader change, Cap n' Proto structures defining messages, an underlying state machine with persistent logs, and various tests and benchmarks. The interfaces are closely integrated, which meant that adding and removing fields in the existing message structure required modifying many function calls and implementation details. Most of the tests and benchmarks also used the Raft protocol's node states and interfaces, and therefore needed to be modified to work with Raft-BFT. Additionally, some Rust packages (e.g. capnp-nonblock and ed25519-dalek) that Raft-rs depended on along with Rust itself had significant modifications in the year since the Raft-rs was last modified, which caused unforeseen problems. All these factors led to unexpectedly amount of extra work, making it difficult to get our implementation working within the short time-frame.

However, it is worth noticing that this in no way invalidates the feasibility of a Rust implementation of our design. We believe that our modification plan on this code base is still feasible and can be completed if more time is allowed. In retrospect, implementing our design from scratch, as we have done in 5.2, would have allowed us to avoid the aforementioned issues. Additionally this implementation likely would've still had better performance than our Python implementation due to the C/C++-level performance of Rust.

5.2. Python

Due to the Rust implementation issues discussed above, we abandoned our Rust implementation, and built an alternative Python implementation from scratch. This implementation is a working example of the transaction mechanism that we theorized above. All of the nodes were run on their own process on the same machine, and used queues for message passing. We were able to build a working implementation for the typical operation of the algorithm (the three stage commits), and were able to benchmark it as shown below.

6. Performance Evaluation

6.1. Rust

Due to the issues in our Rust implementation, we do not have concrete numbers on Raft-BFT's performance in Rust at the time this paper is written. However, we did do some benchmarking to determine how fast the signing algorithm is. Our benchmarking of the ed25519-dalek Rust digital signature package [3] on consumer hardware shows that generating a SHA-512 digest of a given message and signing it takes an average time of 2.2ms, and verifying it takes an average time of 2.7ms. This indicates that digital signature will represent a large portion of the total computation time, and therefore would lead to much higher higher latency than non-BFT Raft implementations. Additionally, as we have the leader do key verification separately, we would spend twice the amount of wall-clock time on key validations as PBFT.

6.2. Python

For the Python performance tests, we can see that there is throughput advantages for putting multiple transactions together. It takes 23.0 ms for our implementation to run one transaction but it only takes 49.7 ms to run ten transactions. However, when there are many transactions to run, for some reason our implementation gets slower. It takes 2.98 seconds to process 1000 transactions but 56.3 seconds to process 10000 transactions. We think this might have to do with our implementation limiting the number of transactions per entry to 1000, but we don't understand why this would make our code scale sub-linearly.

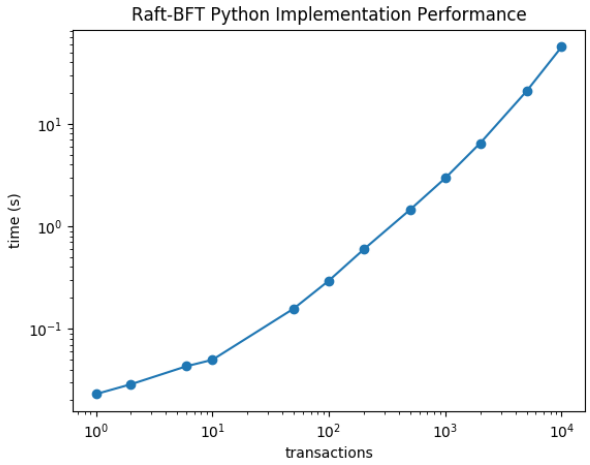

Figure 2. Performance benchmarking results for the Python Raft-BFT implementation.

7. Code

You can find our Rust implementation at : `https://github.com/Jerrymouse23/raft-rs`

And our Python implementation at: `https://github.com/Gamrix/cs244b_proj`

References

[1] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[2] Hoverbear. Raft-rs. `https://github.com/Hoverbear/raft-rs`, 2017.

[3] I. Lovecruft. ed25519-dalek. `https://crates.io/crates/ed25519-dalek`, 2017.

[4] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[5] paenko. Raft-rs. `https://github.com/paenko/raft-rs`, 2017.