# How to make Raft secure?

(CS6500: Term Project Report)

Nischith Shadagopan M N (CS18B102)
Shreesha G Bhat (CS18B103)

June 10, 2022

## Contents

We declare that the work done in this project has not been and will be not used as-is for any other registered course at IITM; if this project code is extended further for any subsequent BTP/MTP project or MS/PhD thesis, this project work will be duly referenced in the reports/theses documents.

# 1 Abstract

We study the problem of making the Raft algorithm secure. Raft is a consensus algorithm which maintains a replicated log across many peers. Understand-ability was the key design consideration when designing Raft. It was intended to be an improvement over the existing algorithm, Paxos, which is quite convoluted to understand and implement. As a result, Raft is not designed with the goal of security and therefore, it is quite easy to come up with attacks on Raft. We try to secure Raft using an application level security protocol, assuming that the underlying network is insecure. We have come up with a security design for Raft which uses a minimum number of security primitives by piggybacking on the properties guaranteed by the existing Raft protocol and keeping in mind its applications. By doing so we expect our approach to be more efficient than a generic approach such as TLS while providing the necessary security guarantees.

# 2 Introduction to Consensus Algorithms and Raft

Distributed Consensus protocols are widely used to implement Replicated State Machines. This method achieves

- *Replication*: Identical copies of a state machine are computed on many distributed servers or nodes.
- *State Machine Safety*: These servers exchange messages to ensure that each update to the state machine is applied in the same order on all the involved nodes. This ensures that clients have a consistent view of the state machine.
- *Fault Tolerance*: Protocols are usually designed in such a manner that the system as a whole can continue to make progress despite the failure of a few nodes.

Replicated State Machines are usually implemented by maintaining a replicated log. The log contains a sequence of operations applied by a state machine. If the underlying distributed protocol ensures that the log maintained by each node contains the same sequence of operations, then we are ensured that each state machine will compute the same sequence of states.

## 2.1 How does Raft work?

Raft [1] is one such consensus protocol designed to maintain a replicated log. The Raft protocol can be split into 3 conceptual sub-problems

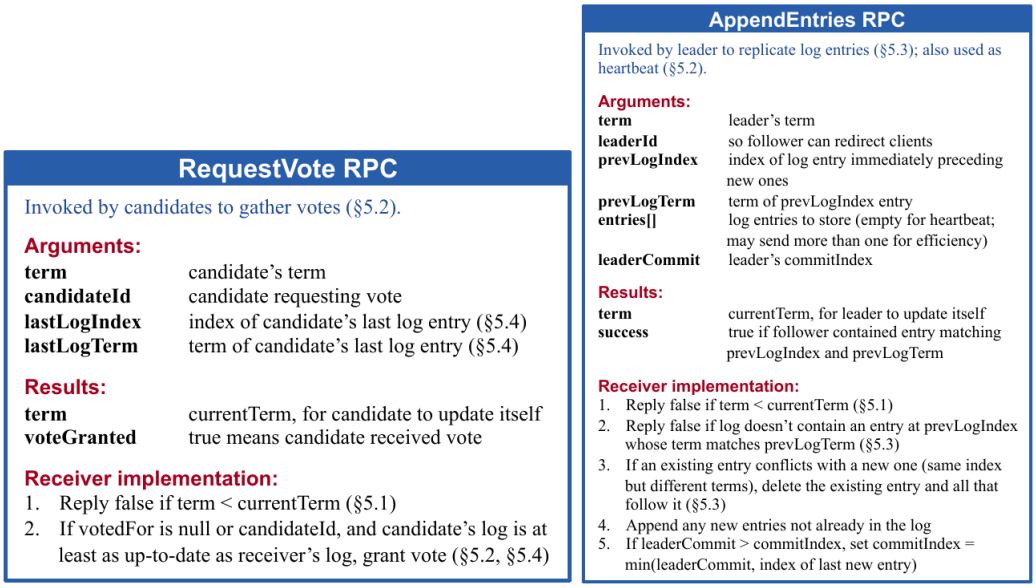
- *Leader Election*: A single leader is elected when the previous leader fails. All information flows from the leader to the followers
- *Log Replication*: The leader accepts updates to the state machine and replicates the same on the followers
- *Safety*: Ensuring that if any server applies a command at a particular index in its log, no other server will apply a different command at the same index to its state machine.

In Raft, time is split into logical *terms* of arbitrary length. In each term, the algorithm ensures that at-most one leader will be elected. Leader election works by majority voting, nodes request for votes and the node that receives votes from a majority of the peers becomes leader for that term. Once a node is elected as the leader, it periodically sends messages called `AppendEntries` (which are either empty, or contain new log entries sent by the client) to the rest of the peers. The rest of

the nodes are considered as follower nodes who passively receive these `AppendEntries`, process these messages, add any entries to their copy of the log if necessary and reply back to the leader with feedback. These follower nodes are withheld from starting new elections as long as they receive these periodic messages from the leader. If a follower does not receive any communication from a leader within a randomized timeout period, the node converts to the candidate state and starts a new election for the next term. Part of the state for each node is persistent i.e. changes to these states are written to the disk before being communicated with any other peer. On failure and restart, the persistent state is used to bring back the server. In general, a set of $n$ raft peers can process client requests as long as $> n/2$ nodes are alive. In this way, Raft maintains a distributed log while also maintaining enough redundancy to tolerate multiple node failures.

### 2.2 Raft RPCs

Raft peers communicate via RPCs or Remote Procedure Calls. The Raft specification describes 2 RPCs, `AppendEntries` RPC and the `RequestVote` RPC.



**RequestVote RPC**

Invoked by candidates to gather votes (§5.2).

**Arguments:**

| | |
|---|---|
| **term** | candidate's term |
| **candidateId** | candidate requesting vote |
| **lastLogIndex** | index of candidate's last log entry (§5.4) |
| **lastLogTerm** | term of candidate's last log entry (§5.4) |

**Results:**

| | |
|---|---|
| **term** | currentTerm, for candidate to update itself |
| **voteGranted** | true means candidate received vote |

**Receiver implementation:**

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

**AppendEntries RPC**

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

**Arguments:**

| | |
|---|---|
| **term** | leader's term |
| **leaderId** | so follower can redirect clients |
| **prevLogIndex** | index of log entry immediately preceding new ones |
| **prevLogTerm** | term of prevLogIndex entry |
| **entries[]** | log entries to store (empty for heartbeat; may send more than one for efficiency) |
| **leaderCommit** | leader's commitIndex |

**Results:**

| | |
|---|---|
| **term** | currentTerm, for leader to update itself |
| **success** | true if follower contained entry matching prevLogIndex and prevLogTerm |

**Receiver implementation:**

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

Figure 1: The fields of the `RequestVote` and `AppendEntries` RPC argument and reply structures as described in the Raft specification

RPCs essentially work like function calls across machines. When calling an RPC, the sender sends an argument structure and receives a reply structure. These argument and reply are transmitted over the network.

The `RequestVote` RPC is used by a candidate to request another peer for votes during an election. The `AppendEntries` RPC is used by the leader to direct a follower to append new log entries to its local copy of the log and update the commit index if necessary.

## 3 Threat Model

- The end systems running the Raft algorithm are assumed to be secure (secure hardware and secure Operating System). Therefore, honest Raft peers trust each other. This means that we do not consider attacks that compromise either the hardware or the Operating System

and thereby gain control of the execution of one of the Raft peers. The end systems running the Raft algorithm are assumed to be resistant to such attacks. One can practically achieve this by running the algorithms in a Sandbox environment, Enclaves or user space isolation techniques such as Intel SGX

- The underlying network used by the systems is insecure. This assumption falls in line with Rafts original assumptions about the network as well. For example, the original Raft protocol can handle message re-transmissions as well as events such as network partitions where groups of peers are completely isolated from one another. Even under such conditions, the original Raft protocol meets its guarantees.

  We extend this assumption by considering malicious attackers that can potentially eavesdrop, modify, replay messages as well as try to impersonate raft peers to break the guarantees provided by the protocol.

# 4 Attacks on Raft: Is the Raft Protocol insecure under the given Threat Model?

The Raft protocol does not address any security concerns. Here are some of the attacks that are possible

- Messages are sent in plaintext, therefore client data is visible to an eavesdropping adversary.
- Like most other consensus protocols, Raft makes use of majority voting to make progress. Furthermore, Raft does not perform any message content or origin authentication, therefore, impersonation attacks are particularly potent. For example, a malicious adversary could impersonate one (or more) of the servers and tamper with the leader election process by voting multiple times for different servers and thereby breaking the guarantee that there is at most one leader in every term.
- Messages can be similarly modified without detection, which can result in the wrong client operations being registered on the log and hence unexpected states in the state machine. Similarly, one can change components of the raft RPCs such as `PrevLogIndex`, `PrevLogTerm`, `LeaderCommit` etc which will result in a compromise of the guarantees promised by Raft.

As we can see above, it is quite easy to come up with attacks on raft. *However, it is important to note that Raft is inherently resistant to pure replay attacks*. The Raft RPCs such as `RequestVote` and `AppendEntries` are *idempotent*, which means that an RPC already processed earlier will not cause any change to the state of a Raft peer if presented again. Therefore, one need not worry about using nonces, timestamps or sequence numbers to secure Raft against Replay attacks. This is an advantage of using and designing custom application layer security protocols which incorporate and use the properties of the underlying application protocol in its design, as opposed to a catch-all, generic approach like TLS which can use many redundant security primitives.

# 5 Securing Raft

## 5.1 Key Distribution: Discussion

To secure Raft communication, we firstly need to ensure that each Raft peer sends RPC arguments and replies that are sent encrypted so that contents are obfuscated. Furthermore, one must

authenticate the argument and reply structures so that a 3rd party cannot impersonate and inject fake messages or modify existing messages meaningfully. To ensure that the encrypted and authenticated structures sent by a Raft peer can be successfully decrypted and verified by the receiver, we need to setup cryptographic keys.

Lets think about setting up asymmetric keys among the peers. One way to do this is to have trusted third party certificates. The problem with this is the authenticated distribution of certificates. Normally this problem is solved by publishing the certificates on a trusted public domain. But in our case we can do something simpler. In applications where Raft is used, all the peers are owned by a single entity/organisation. Therefore the entity can distribute the public keys among the peers in a secure channel outside the network. This method is easier as well as more secure than maintaining a secure website for public keys. In this way the public keys are distributed among the peers.

However, asymmetric keys can only be used for a short duration as the message size is limited and it is computationally expensive. Raft messages can be quite big and hence we cannot rely on asymmetric cryptography for bulk data encryption. We need to set up symmetric keys. One might ask why we cannot distribute symmetric keys the same way we setup public keys. In general, it is harder to distribute symmetric keys as confidentiality and integrity needs to be maintained whereas with public keys, only integrity needs to be maintained. Therefore we need some other way to setup symmetric keys.

Raft sessions can run for a long time, therefore it is insecure to use the same symmetric keys throughout. There are many issues with using the same key for a long time. For example, the longer a key is used, the adversary can collect more ciphertext to perform cryptanalysis and can try to come up with ciphertext only attacks. Also many cryptographic primitives (encryption, decryption) use a random IV and rely on the fact that the IV and key pair must not repeat during the course of the usage of the key. When only one key is being used, the probability of an IV being repeated is higher making the scheme less secure. Since IV is public information, the attacker can keep track of it to identify when it repeats and carry out an attack.

Lastly, it is better to have a shared symmetric key among all the peers than having a symmetric key for each pair of peers. Having a key for each pair is not scalable as the number of keys increases quite fast and also it reduces efficiency as the leader has to encrypt the same message with as many keys as the number of peers. Therefore we need a common shared key. Furthermore, when the key is changed, it needs to be eventually reflected to and shared securely with all the nodes. This task is non trivial as raft peers can go down and restart throughout the run of the algorithm. If a peer is down for a long time and comes back up, it might not know the current key being used and hence it cannot continue executing the Raft algorithm.

### 5.2 Key Change

To change the shared symmetric key under use, and reflect the change across all the peers, we piggyback on the strong consistency property of the log provided by Raft itself. Consider the Raft log maintained by each node to be an array with 0 indexing. Raft ensures that when an entry is added to the log at index $i$ by the leader and replicated at index $i$ across a majority of the nodes, this entry gets "committed" i.e. when building its local copy of the state machine, no raft peer will ever apply a different log entry at index $i$. Therefore, Raft ensures that committed entries are present in the same order in the log of every node.

Our idea is to record key change as a log entry i.e. when the current leader node decides to change the key, it tags the key along with the application data sent to be appended to the log by the client. This way Raft properties ensure that all peers will eventually change to the new key at

this log entry, if this log entry gets committed. We also decided to tie the key change operation to the log entry index, for example the key changes every 100 log entries. This way if a system was down for a long time and has missed log entry (and potentially key change) updates and comes back online, when it replies to the leader in the first `AppendEntries` message it receives, the leader can see the reply structure to determine the largest index upto which the logs of the leader and the lagging follower match and thereby use the latest key the lagging follower has recorded on its log (also present on the leaders log due to Raft consistency properties) to send the new log entries. Therefore, any system which becomes the leader will initiate key change if the log index is appropriate. No other log entry is committed till key change entry is committed. This way we are ensuring that the key is changed periodically. Also we are ensuring a soft limit on the number of log entries that are encrypted using a given key which is roughly the product of the key change frequency (in log indices) multiplied by the number of raft peers. This key change frequency can thus be appropriately set.

Till we elect the first leader which can choose a new key, asymmetric keys are used for encryption and authentication. Also the contents of the argument and reply structures apart from the log entries themselves are always encrypted and signed with asymmetric keys as this should be readable by all peers. This is fine as the header is small in size. If a peer was down for a long time and came back up online it should still be able to read the header to realise that it is lagging behind. Also the leader should be able to read its header to identify it is lagging behind. For similar reasons, the `RequestVote` RPC arguments and reply structures are also encrypted and signed using the asymmetric keys shared.

## 5.3 Example Illustration

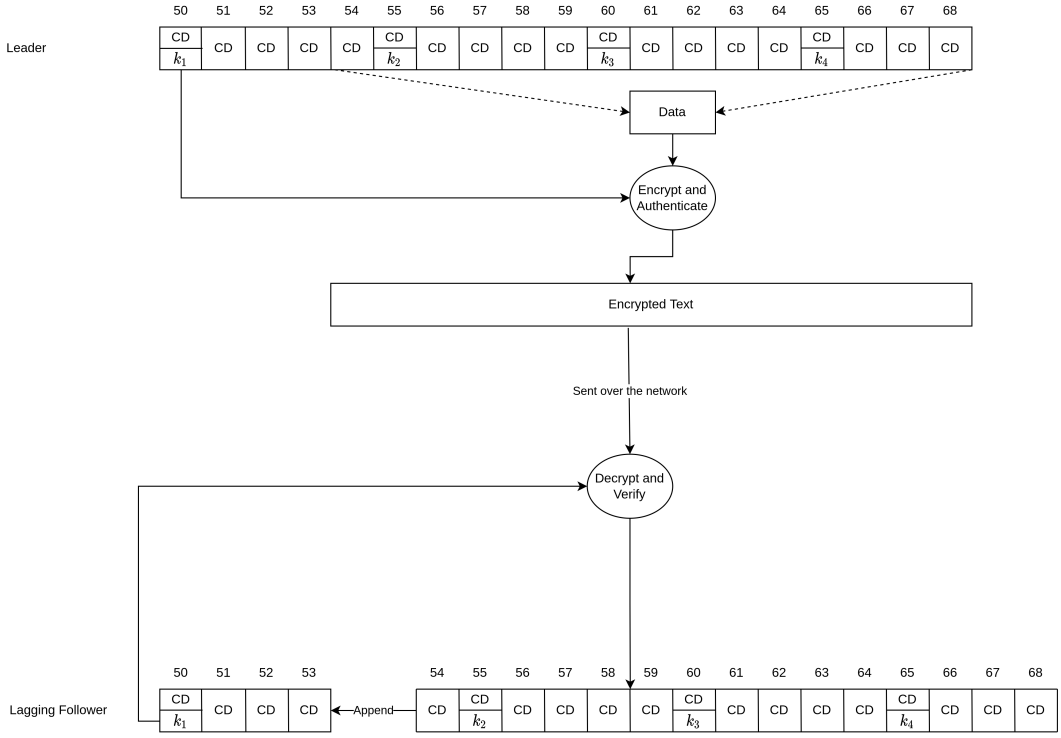



Figure 2: CD refers to client data while $k_i$ represent the different keys used. Assume that we have configured the key change to occur once every 5 log entries. The figure illustrates the procedure by which the current leader sends the updated log to a follower who has been lagging from a while. The latest key present in the log of the follower is also present in the log of the leader. This key is used to encrypt the new log entries being sent by the leader including the newer keys. The encrypted and authenticated data can then be decrypted and verified at the followers end because it also has the same key.

## 5.4 How is each RPC argument and reply structure handled?

| RPC | Argument Structure | Reply Structure |
|---|---|---|
| `AppendEntries` | Log Entries encrypted and authenticated using the symmetric key & change methodology described above, other fields encrypted and signed using the pre-shared public-private key information | Encrypted and signed using the pre-shared public-private key information |
| `RequestVote` | Encrypted and signed using the pre-shared public-private key information | Encrypted and signed using the pre-shared public-private key information |

Table 1: Table describing how the various Argument and Reply structures should be modified to provide confidentiality, integrity and authentication

# 6 Implementation Details

## 6.1 Programming Language used and Development Environment

We implemented the basic Raft protocol in Golang based on the code template developed at MIT for the 6.824 Distributed Computing course. Golang is particularly suited to writing multi-threaded code as concurrency is built into the language in the form of Go routines.

Golang also provides the crypto package to access existing implementations of cryptographic algorithms

## 6.2 Code

We have implemented the naive Raft protocol based on the specification provided in the original Raft paper [1]. We have also implemented the discussed application level security modifications to Raft with a few simplifications described below

### 6.2.1 Simplifications

Although the idea presented to secure raft above works in theory, for a practical implementation, we decided to make the following changes.

- We eliminate the use of public keys due to lack of time and to provide a first working proof-of-concept approximate version on time. Instead of using pre-shared public keys which will be used to generate the first private key used for bulk data encryption, we directly pre-share the first private key to all the raft peers. This key is appended as the first dummy entry in the log of all the nodes. This essentially kickstarts the process.
- Instead of using the public-private key pairs to encrypt and sign the non log entry fields of the RPC structures, we use another pre-shared asymmetric key used uniquely to encrypt and authenticate the contents.
- The AES-256 GCM algorithm in AEAD mode is used to perform authenticated encryption

The above modifications do not significantly affect the security provided, but involves sharing 2 private keys to all the raft peers before hand i.e. (1) The first key $k_1$ used for bulk data encryption provided in the log of all the peers beforehand and (2) A key $k_2$ used to encrypt and authenticate the non log entry fields of the RPCs. Under these simplifications, the table 1 becomes

| RPC | Argument Structure | Reply Structure |
|---|---|---|
| `AppendEntries` | Log Entries encrypted and authenticated using the symmetric key & change methodology described above kick-started using pre-shared symmetric key $k_1$, other fields encrypted and authenticated using the pre-shared symmetric key $k_2$ | Encrypted and authenticated using the pre-shared symmetric key $k_2$ |
| `RequestVote` | Encrypted and authenticated using the pre-shared symmetric key $k_2$ | Encrypted and authenticated using the pre-shared symmetric key $k_2$ |

Table 2: Table describing how the various Argument and Reply structures are modified to provide confidentiality, integrity and authentication in our implementation

# 7 Testing and Performance Study

## 7.1 Testing

The 6.824 Distributed Computing course provides test cases to stress test implementations of Raft which is assignment 2. These tests are divided into 3 components, 2A, 2B and 2C. The set of tests in 2A test whether the basic leader election property is still satisfied under various corner cases involving random as well as targeted failures. Similarly, 2B tests whether consensus is reached on a few basic test cases. 2C does intensive corner case testing and a few other test cases described in the original Raft paper [1], 2C also has a stringent time limit to ensure that the fast log-backtracking optimization mentioned in the Raft paper is implemented. *Both our original Raft implementation as well as the modified Secure Raft pass all the testcases provided consistently.*

## 7.2 Performance Study

In this section, we aim to study the performance implications of the implemented security modifications to Raft. *Note that the MIT framework does not implement the RPC package over TCP but instead simulates a network using channels. But this is an invariant in the comparision, as the same simulated network conditions are present in both the original Raft implementation and Secure Raft*. We run the full test suite for MIT 6.824 Assignment 2 on both the modified Secure Raft implementation as well as the original insecure implementation over 32 runs. We observed that the time taken was not markedly different. Both sets completed in around 13 minutes of real time. Measurements were done using `time` command on Ubuntu when running the test a total of 32 times run 8 in parallel.


Figure 3: Time taken to run the tests on the Original Raft implementation 32 times when run 8 in parallel


Figure 4: Time taken to run the tests on Secure Raft 32 times when run 8 in parallel

From this we see that the cost of securing Raft in terms of performance is very little to non-existant. Therefore, our application layer security protocol performs quite well. On the other hand, we expect a TLS based secure Raft implementation to take much longer time. Our hypothesis is based on the observation that the TLS connections do not persist over machine failures. Therefore, when failures occur frequently (as is the case in many of the test cases), the entire TLS handshake has to be re-done. These costs accumulate over time. Furthermore, using TLS, we have no option but to establish a separate key block and session/connection for each pair of communicating peers. This results in a lot of memory utilization as well and is not scalable.

# 8 Conclusion

We learnt and implemented the Raft protocol. We analysed the Raft protocol for security flaws, under a fixed threat model, based on the security principles we have learnt in class and modified the Raft protocol to make it tolerant to the attacks found. This required application of the secure designs we have discussed in class and also some modifications based on the understanding of Raft.

The CS6500 course provided a basic understanding of the security primitives such as encryption, message authentication etc. We also studied how Network Security can be considered at various layers of the protocol stack. However, this course project provided a unique experience where we could use many of the intuitions we picked up over the course. Firstly, we did not find any literature that discussed "Securing the Raft protocol" which we thought was very strange as security can be of very high concern in applications where Raft is used as the underlying replication and fault tolerance layer. Therefore, we were at the unique position of having to figure out most of the issues and problems by ourselves. We very much enjoyed the process of brainstorming ideas and coming up with better security designs. While we are aware that this work might not be perfect, we feel it definitely holds some value and could be extended and analyzed more formally.

# References

[1] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319. USENIX Association, 2014.