# Raft Implementation Technical Report

Debajit Chakraborty, Hansin Ahuja

December 13, 2024

## 1 Introduction

RAFT is a consensus algorithm enabling server clusters to maintain a replicated log and agree on state changes. Through leader election, servers can transition from followers to candidates and ultimately to leaders with majority support. The leader manages log replication by accepting requests and synchronizing followers, while safety rules ensure consistency by preserving committed entries and requiring up-to-date logs for leadership.

This report outlines our implementation of Raft. Key features of our design include: (1) implementation in Python, (2) message passing via gRPCs and (3) parallelism using Python's threading and concurrent modules. This report outlines our design choices, frontend and backend design, and testing outcomes.

## 2 Background and design choices

Distributed systems require robust consensus mechanisms to maintain consistency across multiple servers. RAFT is a consensus algorithm enabling server clusters to maintain a replicated log and agree on state changes. Through leader election, servers can transition from followers to candidates and ultimately to leaders with majority support. The leader manages log replication by accepting requests and synchronizing followers, while safety rules ensure consistency by preserving committed entries and requiring up-to-date logs for leadership. Our implementation leverages Python for its rapid prototyping capabilities and extensive library support, along with gRPC for efficient, type-safe message passing between components.

Our lectures have discussed Raft in great detail, and we settle on the following design choices to best fit our project's requirements.

1. **Programming language:** We have elected to implement Raft in Python.
   (a) Rapid prototyping: Python's simplicity and extensive library ecosystem allow for quick development and iteration.
   (b) Concurrency support: Python's threading and concurrent modules provide robust tools for handling parallel operations, which is crucial for implementing Raft's distributed nature.
   (c) Cross-platform compatibility: Python's portability ensures our implementation can run on various operating systems with minimal modifications.
2. **Messaging:** As suggested in the requirements document, message passing between servers, clients, and the frontend will be implemented using gRPC. We believe gRPC is an excellent choice for our Raft implementation due to the following reasons:
   (a) Performance: gRPC uses Protocol Buffers for serialization, resulting in smaller message sizes and faster processing compared to text-based protocols like REST.

(b) Language agnostic: While we're using Python for our implementation, gRPC's language-agnostic nature allows for easy integration with potential future components written in other languages.

(c) Strong typing: gRPC's use of Protocol Buffers ensures type safety, reducing the likelihood of runtime errors due to incorrect message formats.

## 3 Architecture

Figure 1 shows the architectural flow of our implementation

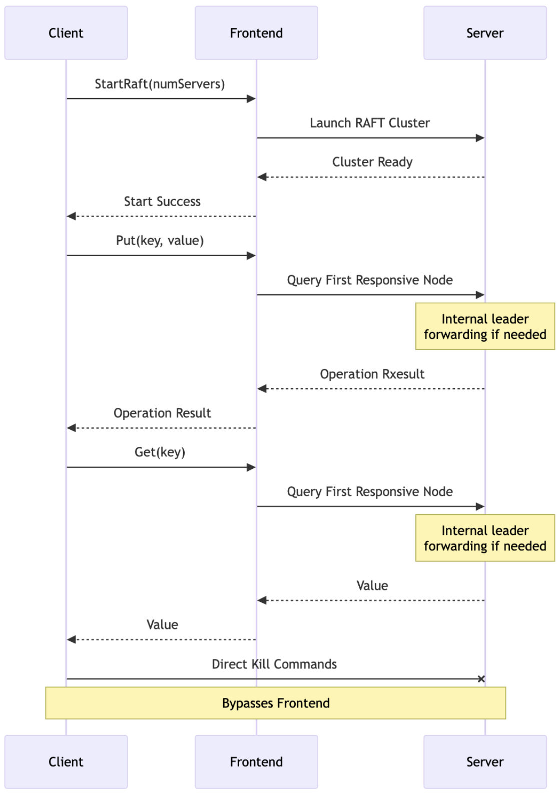

Figure 1: Architectural flow

The following sections detail our low-level design and methodology, with **challenges** attached to each section.

## 4 Frontend Service

The front-end service acts as a proxy between clients and the RAFT cluster. Key responsibilities include

### 4.1 Server Launch Implementation (StartRaft)

The StartRaft function is responsible for initializing and launching multiple RAFT servers in a cluster configuration. Some of the challenges we faced here, was trying to open the servers without terminal logging since we were testing on a virtual machine. Another challenge we faced, was trying to correctly name the server process as raftserver since pkill commands used later in testing often failed to kill the actual process leaving behind zombie processes.

### 4.1.1 Pre-launch setup

With every run of the algorithm, we ensure to clean the existing persistent memory directory. In addition, based on the request of the client we update a global configuration file and finally we proceed to create a address mapping from server id to the actual IP addresses.

```
config = configparser.ConfigParser()
config.read(CONFIG_FILE)
config.set('Servers', 'active', ",".join(map(str, range(num_servers))))
with open(CONFIG_FILE, "w+") as configfile:
    config.write(configfile)

for i in range(num_servers):
    server_id = i
    server_port = BASE_SERVER_PORT + server_id
    self.address_map[server_id] = f"{self.base_address}:{server_port}"
```

### 4.1.2 Server Process Creation

For each server, we are creating a dedicated log file to track any output and errors after launching the server script. In order to name the python process as `raftserver1, raftserver2`, etc. we use the `exec -a` command which ensures the python scripts running are mapped to the correct names as required. Finally, using the `Popen` function we launch multiple servers.

One interesting thing we observed was that if we simply named the process without explicitly handling the termination process, the processes were not killed properly. Infact, despite being zombie processes they were live and kept listening on their port which lead to false consensus. This was done using this command `signal.signal(signal.SIGTERM, signal.SIG_DFL)`

```
python_cmd = (
    'import os, sys, signal; '
    'signal.signal(signal.SIGTERM, signal.SIG_DFL); '
    f'import server; server.run_server({server_id})'
)

cmd = [
    'bash',
    '-c',
    f'cd {current_dir} && exec -a raftserver{server_id+1} python3 -c \'{
       python_cmd}\''
]
```

## 4.2 Get Live Stub

After creating the servers, we need to have a function for the various frontend methods to communicate with the servers which happens through ports as mentioned in the document `9001, 9002, 9003,` etc. We use our global configuration file to establish the address map. Then we proceed to establish gRPC channels. Whichever is the first live server, we return that corresponding stub to the function requesting a stub. Some of the challenges we faced in `getStub()` was establishing a correct connection with the server nodes, especially if were not clearing any existing processes using that port.

```
server_id = int(active_servers[i])
server_address = f'{base_address}:{str(base_port + server_id)}'
channel = grpc.insecure_channel(server_address)
stub = raft_pb2_grpc.KeyValueStoreStub(channel)
```

## 4.3 Put Operation

To handle concurrent requests, we implement our Put operations using a ThreadPoolExecutor to ensure multiple operations are happening in parallel.

```
def Put(self, request, context):
    return self.executor.submit(self.put_task, request, context).result()
```

The threads call a function called put_task which redirects our requests to the main server Put function. This is done after requesting the stub as described by the get_stub function.

```
try:
    server_request = raft_pb2.KeyValue(
        key=request.key,
        value=request.value
    )
    # calling actual server Put
    response = leader_stub.Put(server_request)
    reply = raft_pb2.Reply(
        wrongLeader=not response.success,
        error="" if response.success else "Put failed"
    )
    # if Put fails, we did not reach consensus
    if response.success == False:
        context.abort(grpc.StatusCode.FAILED_PRECONDITION, "Did not reach
            consensus")
    return reply
except Exception as e:
    self.current_leader = None
    return raft_pb2.Reply(wrongLeader=True, error=str(e))
```

Here, the primary challenge was that if the Put function in server fails, it means that we did not acheive consensus for the key value store operation and thus we return an error back to the client. A small caveat is that, in our server Put function we also introduce a wait of 0.9 seconds before querying for consistency.

## 4.4 Get Operation

Similar to the Put operation, we also use a ThreadPoolExecutor to ensure parallelism.

```
def Get(self, request, context):
    return self.executor.submit(self.get_task, request).result()
```

To make our get operations more robust, we also introduce retry logic in our get_task code. Similar to the put_task function, we query our backend for the value of the key as requested by the client. On successful fetch, we return a successful response back to the client. Also for ensuring the consistency is reached before our request is fetched.

```
def get_task(self, request):
    for retry in range(max_retries):
        leader_stub = self.get_stub()
        if not leader_stub:
            return raft_pb2.Reply()
        try:
            server_request = raft_pb2.StringArg(arg=request.key)
            response = leader_stub.Get(server_request)
            if response.value != 'None':
                return raft_pb2.Reply(
                    wrongLeader=False,
                    value=response.value
                )
            if retry < max_retries - 1:
                time.sleep(retry_delay)
                continue
        except Exception as e:
            ...
```

## 5 Server Service

Our implementation of Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger understanding to reduce the number of states under consideration.

### 5.1 Election Protocol Implementation

Each server initiates candidacy by incrementing its term, voting for itself, and requesting votes from other servers in parallel, transitioning to leader if it receives a majority of votes or stepping down to follower if it discovers a higher term. Here, configuring good timeouts was the biggest challenge. Too small a timeout implies that followers may not receive heartbeats from the leader in time before initiating election. Too long a timeout, in the case of leader failure, would lead to followers waiting too long before starting a new election.

#### 5.1.1 Candidate Declaration

On declaring candidacy, the servers starts an atomic role transition to 'candidate'. The term increment is ensured to be monotonic. The candidate starts with a self-vote implementation and starts a thread-safe parallel vote request dispatch.

```
def declare_candidacy(self):
    self.role = 'candidate'
    self.currentTerm += 1
    self.votedFor = self.id
    self.voteRequestCalls = []
    for server_id in ADDRESS.keys():
        if server_id != self.id:
            self.voteRequestCalls.append(Thread(target=self.
                send_vote_request, args=[server_id]))
    for t in self.voteRequestCalls:
        t.start()

    self.start_timer(self.count_votes)
```

#### 5.1.2 Vote Request Protocol

Following from the previous candidacy declaration, this function actually implements the `RequestVoter` method. After the successful collection of votes from other servers, it populates the votes in an array.

```
def send_vote_request(self, server_id):
    if self.role != "candidate":
        return
    try:
        lastLogTerm = 0
        if len(self.log):
            lastLogTerm = self.log[-1]['term']
        request = raft_pb2.RequestVoteRequest()
        response = stub.requestVote(request)
        if response.voteGranted:
            self.votes[server_id] = 1
        elif response.term > self.currentTerm:
            self.votedFor = None
            self.currentTerm = response.term
            ...
            self.start_timer(self.declare_candidacy)
    except Exception as e:
        pass
```

## 5.2 Leader operations

The leader maintains authority by sending periodic heartbeats (empty AppendEntries RPCs) to all followers, replicates new log entries to followers, and manages log consistency by tracking matchIndex and nextIndex for each follower while stepping down if it discovers a higher term.

### 5.2.1 Heartbeat Mechanism

Hearbeats are sent by the leader which are used to check liveness of other servers and acts as a log replication vehicle across servers. It ensures leader commit propagation by calling `AppendEntries`. If the leader finds that its term lesser than any of the responders' terms, it becomes a follower.

```
def send_heartbeat(self, server_id):
    if self.role != 'leader':
                return
    try:
        request = raft_pb2.AppendEntriesRequest(
            term=self.currentTerm,
            leaderId=self.id,
            prevLogIndex=self.nextIndex[server_id]-1,
            prevLogTerm=self.log[self.nextIndex[server_id]-2],
            ...,
        )

        response = stub.appendEntries(request)

        if response.term > self.currentTerm:
            self.votedFor = None
            self.currentTerm = response.term
            self.role = 'follower'
            self.write_persistent_memory()
            self.start_timer(self.declare_candidacy)

        ....

    except Exception as e:
        pass
```

### 5.2.2 Log Append Protocol

`AppendEntries` is crucial to RAFT's design. It is responsible for synchronizing terms across servers. It recognizes a new leader and resets the election timeout and steps down to a "follower" role. It is also responsible for log continuity and also truncates the log at `prevLogIndex` which is responsible to log consistency between leader and followers. Finally it persists the change to stable storage and applies the commit in the order of the leader.

```
def appendEntries(self, request, context):
    if request.term < self.currentTerm:
        return raft_pb2.AppendEntriesResponse(term=self.currentTerm,
            success=False)

    if request.prevLogIndex > len(self.log):
        return raft_pb2.AppendEntriesResponse(term=self.currentTerm,
            success=False)
```

## 5.3 Consensus Protocol Management

The system achieves consensus through majority-based commitment where the leader only commits entries after confirming replication on a majority of servers, applies committed entries to its state machine in order, and ensures all followers eventually replicate and apply the same log entries in the same order.

### 5.3.1 Vote Counting Mechanism

This function calculates the votes after collecting from all the threads decides if a quorum has been achieved. If the candidate loses the election, it reverts back to the follower state and schedules a new election attempt. On winning the election, it becomes the leader and begins tracking committed entries for each follower. It also starts sending heartbeats to prevent new elections and establish authority.

```
def count_votes(self):
    for t in self.voteRequestCalls:
        t.join(0)

    min_votes = math.ceil(len(ADDRESS) / 2)
    if len(ADDRESS) % 2 == 0:
        min_votes += 1

    if sum(self.votes) < min_votes:
        self.role = 'follower'
        self.randomTimeout = random.uniform(MIN_RANDOM_TIMEOUT,
            MAX_RANDOM_TIMEOUT)
        self.start_timer(self.declare_candidacy)
    else:
        self.role = 'leader'
        self.leaderId = self.id
        self.randomTimeout = LEADER_TIMEOUT
        self.matchIndex = [0 for i in range(len(ADDRESS))]
        self.nextIndex = [(len(self.log)+1) for i in range(len(ADDRESS))]
        self.voteRequestCalls = []
        for server_id in ADDRESS.keys():
            if server_id !=             self.id:self.voteRequestCalls.
                append(Thread(target=self.send_heartbeat, args=[server_id])
        for t in self.voteRequestCalls:
            t.start()

        self.start_timer(self.update_kv_store)
```

### 5.3.2 State Machine Replication

This method is used by the leader to update its own indices and check logs with other servers to see if majority has been reached. It advances `commitIndex` only if a majority consensus has been reached. Finally, it applies the state machine and starts new heartbeats.

```
def update_kv_store(self):
    if self.role != "leader":
        return
    for t in self.voteRequestCalls:
        t.join(0)
    self.nextIndex[self.id] = len(self.log)+1
    self.matchIndex[self.id] = len(self.log)
    successful_commits = 0
    for i in range(len(self.matchIndex)):
        if self.matchIndex[i] - 1 >= self.commitIndex:
            successful_commits += 1

    if successful_commits > math.floor(len(self.matchIndex) / 2):
        self.commitIndex += 1

    while self.lastApplied < self.commitIndex:
        self.kv_store[self.log[self.lastApplied]['key']] = self.log[self.
            lastApplied]['value']
        self.lastApplied += 1

    self.voteRequestCalls = []
    for server_id in ADDRESS.keys():
        if server_id != self.id:        self.voteRequestCalls.append(
            Thread(target=self.send_heartbeat, args=[server_id]))
    for t in self.voteRequestCalls:
        t.start()
    self.start_timer(self.update_kv_store)
```

### 5.3.3 Intra-Server Communication

This function deals with the requirement as outlined in the design document for the RAFT implementation. We create a socket with specific source port and use gRPC's insecure port channel to create a channel with the destination port which is 900x. The source port for each server is mapped as `BASE_SOURCE_PORT + id * num_servers + target_id`. This implementation uses unique source ports for each connection and prevents port conflicts in multi-server setup. *We also also include a separate discussion in our results with our observations on Python's gRPC module and its source port handling at the end of our report.*

```
def setup_client_connections(self, me):
number_of_servers = len(ADDRESS)
start_idx = 7001 + me * number_of_servers

for i in range(number_of_servers):
    if i != me:
        port_number = 9001 + i
        source_port = start_idx + i
         try:
            # Create socket with specific source port
            sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            sock.bind(('', start_idx + i))
            custom socket
            channel_options = [
                ('grpc.custom_primary_socket', sock.fileno())
            ]
            channel = grpc.insecure_channel(
                f'localhost:{port_number}',
                options=channel_options
            )
            self.servers_to_clients[i] = channel

        except Exception as e:
            print(f"Failed to connect: {e}")
```

## 5.4 Persistent State Management

We handle server crashes using a persistent memory store management system. We log the most recently committed data to our own logs which are used later by a server during its restart.

```python
def write_persistent_memory(self):
    data = {
        "currentTerm": self.currentTerm,
        "votedFor": self.votedFor,
        "log": self.log
    }
    if not os.path.exists(PERSISTENT_STATE_PATH):
        os.makedirs(PERSISTENT_STATE_PATH)
    with open(os.path.join(PERSISTENT_STATE_PATH, f'server_{SERVER_ID}.
       json'), 'w') as file:
        json.dump(data, file)
```

Python takes significantly longer to write to file than other languages. To overcome this, memory persistence is executed using a background process once the decision to write to file has been made.

## 6 Results and Testing Service

All of the test cases are passing for our implementation of RAFT consensus algorithm (with a caveat for the 3 tests involving network partitions, which we'll explain). We are going to dive briefly into each of the testing functions and explain how our algorithm handles the testing parameters.

### Assumptions

We need to ensure that there are no stray "raftserver" processes in the system before running the tests

### General Observations

`TestStartRaft()` successfully works in our case since our average time to start raft is 0.5 seconds which we have hard-coded to allow for the servers to startup. `GetLastLeader()` also works successfully for our case which queries each of the live server about it's leader status. For the `TestLinearizability()`, initially we were facing issues in getting consensus for our `Put()` requests, within the time limits. However in our server implementation, we introduced a small wait to achieve consensus.

Here is a list of tests that ran successfully for us on our Linux VM:

- `TestStartRaft()`
- `GetLastLeader()`
- `TestLinearizability()`
- `DisconnectMinority()`*
- `DisconnectMajority()`*
- `TestOperations()`
- `TestKillLeader()`
- `KillOneNode()`
- `DisconnectLeader()`*
- `TestLinearizabilityWithKeyValueDuplication()`**

*Some caveats discussed in the next section.

**Our understanding as per a discussion on Ed was that this would not be tested. However, this test passes as well.

## The challenge with network partitions

While implementing our `server.py` we ensured to set a specific source and destination ports to our socket through which we opened a gRPC connection between the servers. Upon investigation, we identified that while we specify a source port on the sender side, the receiver observes the request arriving from a different port. This issue appears specific to Python's gRPC implementation. When creating a channel in Python gRPC, the library abstracts the underlying socket management. The custom socket factory we implemented serves as a suggestion, but the operating system's network stack ultimately determines the source port assignment, which assigns an ephemeral port in the range 49152–65535. In contrast, this behavior does not manifest in Go or C++ (tested using a minimal implementation), where source port assignment behaves as expected. The reason for this is that there is a solution for this already in C++, but still an open issue for Python (`https://github.com/grpc/grpc/issues/29107`).

As a result, the servers continued to communicate even after introducing the relevant iptable rules. So, we implemented our own versions of the 3 tests involving network partitions, which do not use `BlockPort` and `UnBlockPort`. In our modification, we kill the node to be partitioned out and then bring it back up using another API `StartServer` exposed by the frontend. The testing suite provided to us ensures that for any server $S_i$ to be partitioned out, packets are dropped between $S_i$ and $S_j$ for all i != j. We did not identify any communication between $S_i$ and the frontend or server either. Hence, $S_i$ was not communicating with any other port. Our modification ensures complete disconnection from other servers, which aligns with the intended test conditions. We believe this change is functionally equivalent.

We have included the modified testing suite along with our code submission.