UNIVERSITÀ DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato finale in **Ingegneria del Software**

A Study of Two Consensus Algorithms: Raft and Viewstamped Replication

Anno Accademico 2024/2025

Candidato
Francesco Cozzuto
matr. N46004043

Indice

Chapter 1

Introduction

1.1 The Consensus Problem

In order to build fault-tolerant systems, one of the most established approaches is to build them as distributed systems. To ensure that no single machine (node) in the system is a single point of failure, it must be possible for nodes to replace the failed ones. The standard approach to achieve this is through replication algorithms, which maintain consistent copies of the system's state across multiple machines (replicas). By coordinating these replicas, the system can tolerate the failure of individual nodes and continue to process client requests without interruption.

The replication mechanism is only possible if the nodes are able to reach a consensus, as they need to agree on the order of operations.

Protocols that allow multiple nodes of a system to reach consensus are referred to as consensus algorithms.

Consensus is a concept that goes beyond fault-tolerant systems. Solutions to the consensus problem are highly dependent on their application environment. For example, the network may be synchronous or asynchronous, and failed nodes may follow a crash-stop, crash-recovery, or Byzantine (arbitrary-fault) model [1].

In the context of fault-tolerant systems, it is common to assume an asynchronous network such as the Internet and a crash-recovery processing model. Given such constraints, the most established consensus algorithms are Paxos [2] and the more recent Raft [3].

1.2 History of Consensus Algorithms

The first consensus algorithm for nodes operating in asynchronous network and with a crash-recovery model was Viewstamped Replication (VSR), published in 1988 by Oki and Liskov [4]. The algorithm was designed to work with a distributed database, so the replication algorithm was tied to a transaction system.

The following year, 1989, Lamport first submitted "The Part-Time Parliament" [5] where he presented a consensus algorithm. The paper presented the algorithm as an allegory of the parliament of Paxos, a Greek island where participants often left the island. The paper was

published in 1998 after being rejected twice due to the indirect nature of the explanation.

Both VSR and Paxos, which were developed independently, built on the work of R. H. Thomas [6] and D. K. Gifford [7] who first identified the quorum intersection property. This allowed a new class of replication algorithms capable of handling up to f failures where $2f + 1$ is the size of the cluster.

In 1985, Fischer et al. demonstrated that it is impossible to ensure consensus in an asynchronous network if even just one node may fail [8]. Paxos and VSR solved this problem by relaxing the asynchronous network assumption by using timeouts to detect node failures, effectively enforcing an upper bound on message transmission times.

The topic of consensus remained a mostly theoretical subject until around 2000-2015, when the need to build more scalable systems made the topic relevant in the industry. Paxos was considered particularly complex, so Lamport published a simplified explanation of the protocol [2]. Paxos subsequently became the standard solution for consensus.

A revised version of VSR was also published in 2012 by Liskov [9], presenting a more general variation of the algorithm. Lastly, in 2014 D. Ongaro and J. Ousterhout presented Raft [3], a new consensus algorithm designed with understandability and intuitive correctness in mind. Raft rapidly gained popularity becoming the reference consen-

sus algorithm for new projects, especially in open-source.

Despite the chronological gap between their inception, Raft and VSR are remarkably similar compared to Paxos. Rather than acting as generalized consensus protocols, both are designed to solve the specific problem of replication.

Chapter 2

Protocols

2.1 State Machine Replication

Replication and consensus are related but solve different problems. Consensus is the mechanism by which nodes agree on a single value, while replication utilizes consensus to consistently duplicate a single state across multiple machines. Replication algorithms are usually categorized as passive and active. Since VSR and Raft are active replication algorithms we will focus on those.

In active replication, operations are sent to and executed by all replicas. To guarantee that results do not diverge, nodes must behave as deterministic state machines, where the outcome of each operation depends solely on that operation and the history of previously applied operations. As long as the same operations are applied in the same

order, their states will be synchronized.

2.2 Quorum Intersection Property

Consensus algorithms leverage the quorum intersection property to ensure the overall system behaves in a cohesive way in the face of node or network failures. At its core, this property guarantees that any two majorities of nodes within a system will always share at least one common node.

This overlapping node guarantee is essential for resolving two critical failure scenarios:

- **The "Split-Brain" Problem:** Network partitions can sever communication between different groups of nodes. If these isolated groups were allowed to continue normal operations independently, their states would diverge, causing inconsistencies. The quorum intersection property prevents this as at most one partition can ever hold a majority. The system operates under the assumption that it is better to not respond at all than to respond inconsistently, ensuring only the partition with a majority can proceed.

- **The Lost Updates Problem:** Due to node unavailability, it is common for some nodes to miss updates and fall behind. If the nodes holding the most recent state crash, the remaining

cluster might proceed without knowledge of the latest updates, leading to state divergence. The quorum intersection property solves this: if two updates are successfully applied, they must have been agreed upon by majorities which must have at least one common node that is aware of both updates.

Algorithms that leverage the quorum intersection property can function correctly as long as a majority of the nodes remain live, effectively tolerating any minority of nodes failing:

- In a cluster sized $2f + 1$ nodes, the system can tolerate up to f simultaneous faults.
- Increasing the cluster size to $2f + 2$ nodes does not increase the maximum number of faults the system can tolerate.

2.3 Viewstamped Replication (VSR)

VSR is a replication algorithm based on state machine replication where a single node acts as primary accepting client operations and forwarding them to the other nodes, called backups. If the primary crashes, a different node takes its place through a process called view change. When a node restarts after a crash (be it the primary or a backup), it downloads the current state from other live nodes and goes back to normal operation.

2.3.1 Normal Operation

During normal operation, clients send requests to the primary which adds them to its in-memory log and forwards them to backups using PREPARE messages. At this time, operations are not yet considered committed, which means it is possible for the system to forget they were received. The leader must first ensure the backups have acknowledged the operations.

When backups receive a PREPARE message, they add the operation to their log and respond to the primary with a PREPARE_OK message. When the primary receives at least f messages, a quorum of $f + 1$ has been formed for it. Any future majority will include at least one node that knows about this change.

The primary marks the log as committed up to that entry and applies those operations to the state machine. Periodically the primary sends COMMIT messages with the number of committed entries in the log to backups, allowing them to synchronize with it by committing them and applying those operations to their version of the state machine.

2.3.2 View Change

Primary or backups may crash due to an error. Upon restarting, nodes enter a recovery state. If the crashed node was the primary, the rest of the cluster must select a new primary to continue operation.

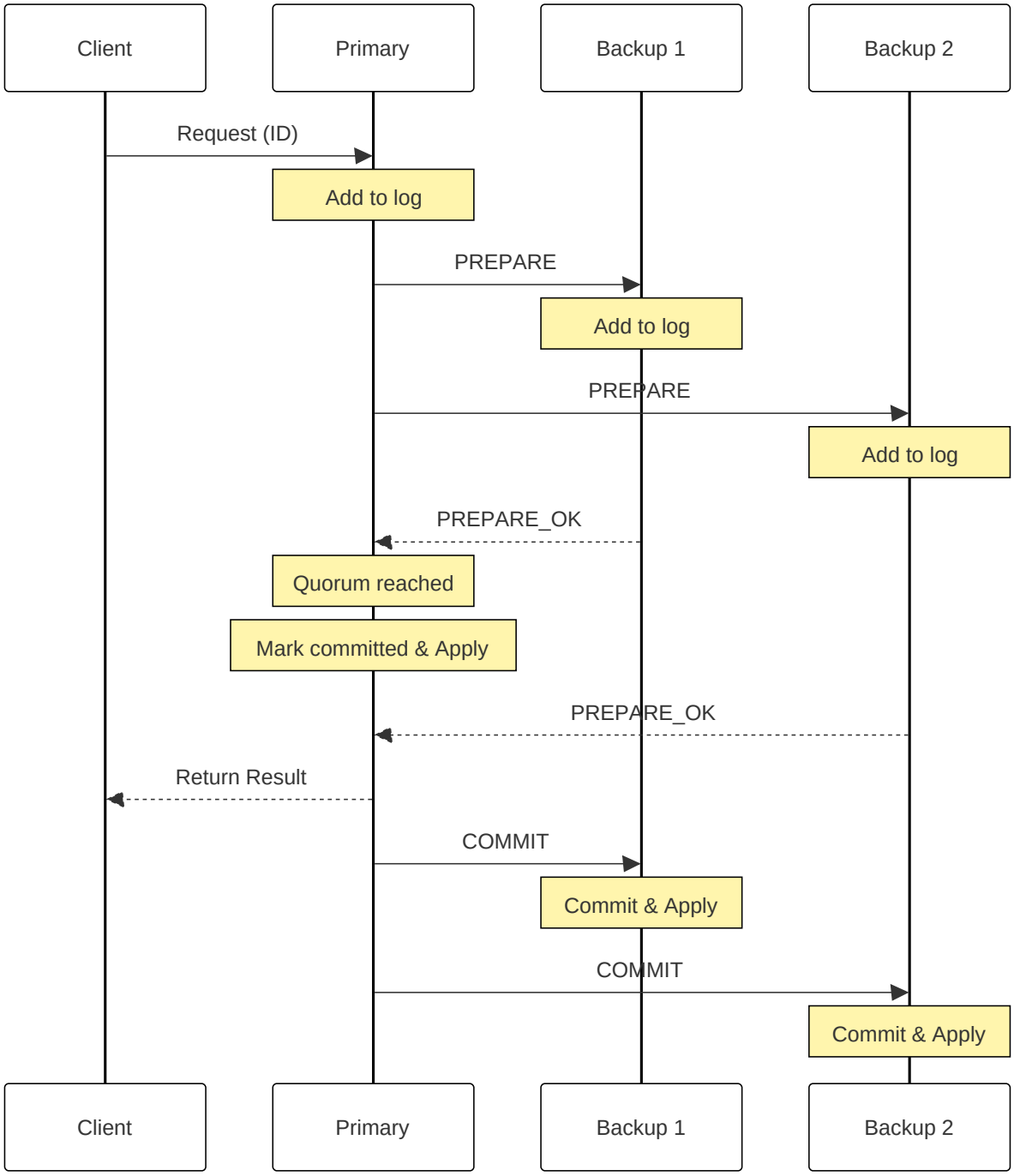

Figure 2.1: Message flow during the normal operation phase of VSR. The primary node forwards client requests using PREPARE messages, waits for a quorum of $f + 1$ nodes, and synchronizes backups using COMMIT messages.

This mechanism, which is referred to as "view change" in VSR, picks primaries in a round-robin fashion. The list of nodes in the cluster, which is set statically, is sorted by their address. When a primary crashes, the next one in the list is selected.

Backups detect the primary is unavailable as they do not receive a COMMIT message before a predefined timeout. When that occurs, they enter the view change phase and send a START_VIEW_CHANGE to other backups to trigger their timeouts. When each node receives f START_VIEW_CHANGE messages, every node knows that a quorum for the view change has been reached.

Nodes then send a DO_VIEW_CHANGE message to the new primary containing their logs and commit numbers. The new primary identifies the most up-to-date commit number and log by comparing its own with the ones it received from the backup nodes. When this process is complete, the new primary sends a START_VIEW message with the new authoritative log. Backups will move forward using this new log.

The new log may contain uncommitted entries. If that is the case, backups immediately send PREPARE_OK messages for them, allowing the primary to commit them.

2.3.3 Recovery

When a node crashes and comes back online, it sends RECOVERY messages to its peers which respond with RECOVERY_RESPONSE messages. The primary also sends the current log alongside the message. When the recovering node receives $f + 1$ responses including the one from the primary, it goes back to normal operation.

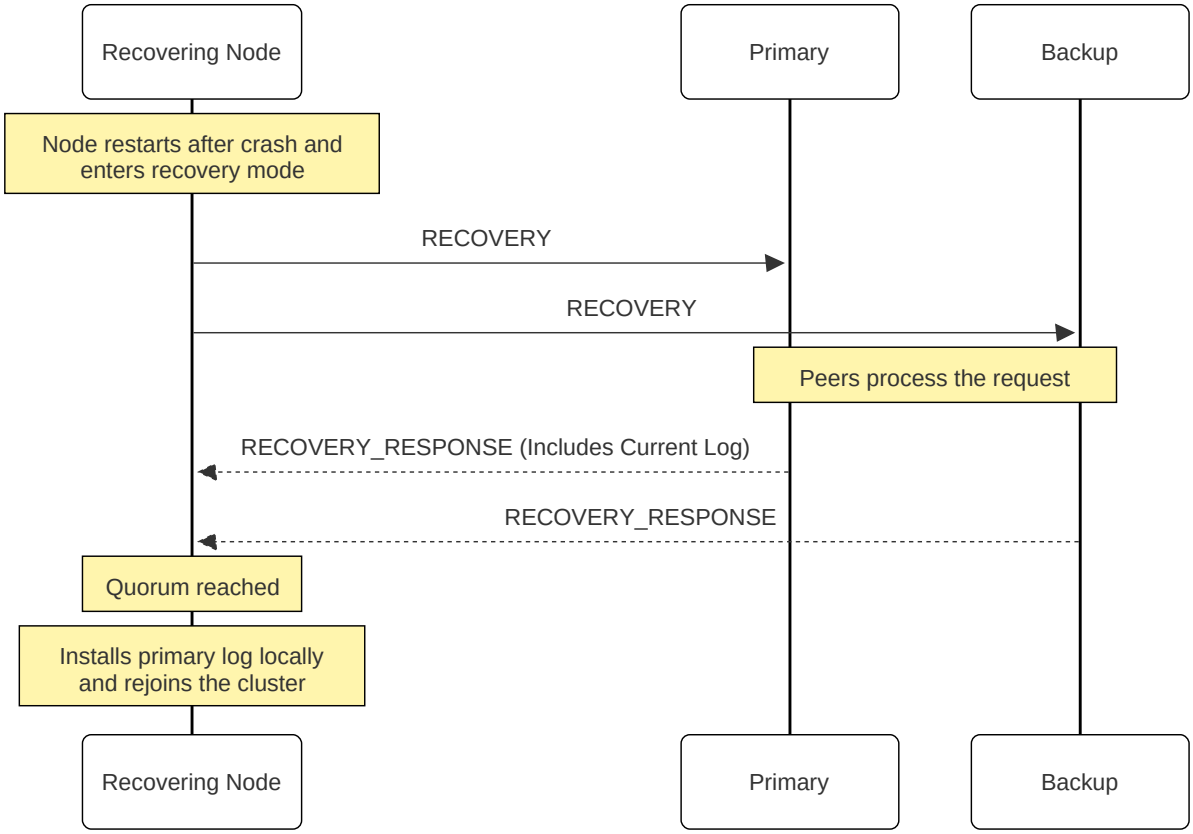

Figure 2.2: Sequence diagram of the VSR recovery phase. Upon restarting from a crash, a node enters the recovery state and broadcasts RECOVERY messages. Once it receives $f + 1$ RECOVERY_RESPONSE messages (including the authoritative log from the primary node) it rebuilds its state and transitions back to normal operation.

2.3.4 Client Table

When a client submits an operation to the system, it must give the request an integer ID. Only one request per client can be in-progress at any given time, and later requests must have a higher ID than previous ones.

The primary node keeps track of client requests using a "client table". This allows the primary to reject duplicate or out-of-order requests to enforce the overall linearizability of the system.

When the primary receives the first request of a client, it adds an entry for that client in the client table containing the client ID and request ID. When the operation is committed, its result is stored in that client table row. When a subsequent request from that client is received, the primary compares the request ID with the previously stored one:

- If the request ID is lower, the request is stale and therefore ignored;
- If the request IDs are the same but no result was saved, the message was duplicated and therefore ignored;
- If the IDs are the same and a result is available, that result is returned to the client;
- If the ID equals the previously stored ID plus 1, it is forwarded to backups, else it is ignored as a previous message was lost on

the network from that client.

2.4 Raft

Similarly to VSR, Raft also consists of a log replication mechanism that uses a leader node to enforce a global ordering of operations. Unlike VSR, leaders are chosen randomly via an "election" process.

2.4.1 Term

In Raft, time is logically organized in "terms" that start with each new election process. Terms are identified by increasing integers, allowing nodes to detect stale messages from previous terms and when they are lagging behind by receiving greater term numbers.

2.4.2 Election

Leaders periodically send heartbeat messages called AppendEntries to replicas in order to keep their authority.

In Raft, a node operates in one of three states: follower, candidate, or leader. When a follower does not receive a heartbeat by a certain time referred to as "election timeout", it starts the election process: it promotes itself from follower to candidate, increments the term, votes for itself, sends RequestVote messages to its peers.

The RequestVote contains the index of the last log entry and the term

it was generated in. If nodes did not vote for anyone else and they find the candidate's log is at least as recent as their own, they vote positively, else they vote negatively. If the candidate receives f or more positive votes, it promotes itself to leader and lets its peers know by sending AppendEntries messages.

It is possible for multiple nodes to promote themselves to candidate simultaneously. The probability of this happening is mitigated by assigning a randomized election time to each node between 150ms and 300ms. Even then, it is still possible for candidates to request votes simultaneously. If no node reaches a quorum, the term ends prematurely allowing a new election. This scenario is referred to as "split-vote".

2.4.3 Log Replication

When a client sends a request to the leader, the leader appends it to its log. The operation is considered as uncommitted and is forwarded to replicas alongside the next AppendEntries heartbeat message. If a quorum of replicas accept the new entry, the leader commits and applies to the state machine that entry and all previous uncommitted ones.

AppendEntries messages also contain the number of currently committed entries, which means replicas will learn about the newly committed entries at the next AppendEntries message, causing them to also com-

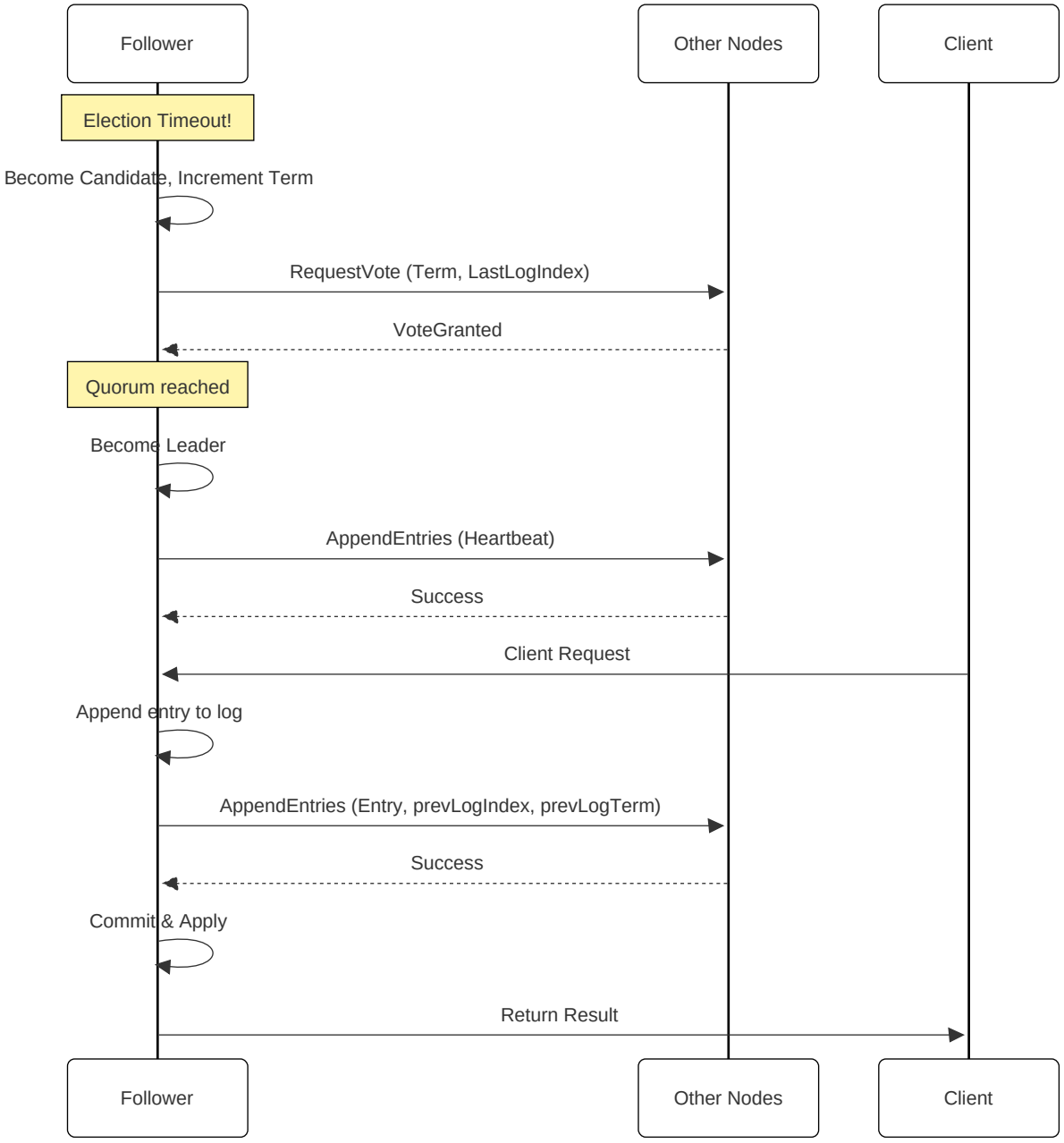

Figure 2.3: Sequence diagram of the Raft election. When a follower's election timeout expires, it transitions to the candidate state and requests votes from its peers. Once it receives a quorum of $f + 1$ votes (including its own), it promotes itself to leader and begins sending periodic AppendEntries heartbeats to maintain its authority and synchronize logs.

mit and apply them to their local state machine.

In general, it is possible for replica logs to diverge from the leader's due to network failures. The log replication mechanism must therefore account and correct such scenarios. This is done using Raft's log matching property: if two log entries have the same index and term number, that entry and all previous ones are identical.

AppendEntries messages contain the index and term of the log entry that precedes the ones being sent. When a replica receives the message, it checks that its last entry matches the index and term. If so, all previous entries match the leader's. If not, the replica rejects the entries by sending a negative message.

Upon being rejected, the leader increases the portion of the log that replica does not know about. The next AppendEntries message will contain an additional entry. This process continues until the last entry that is not diverged between leader and replica is found, so that the replica can then accept the entries and synchronize with the leader.

It is important to note that unlike VSR the log must be stored on disk as Raft does not offer a mechanism for crashed nodes to acquire their state from its peers.

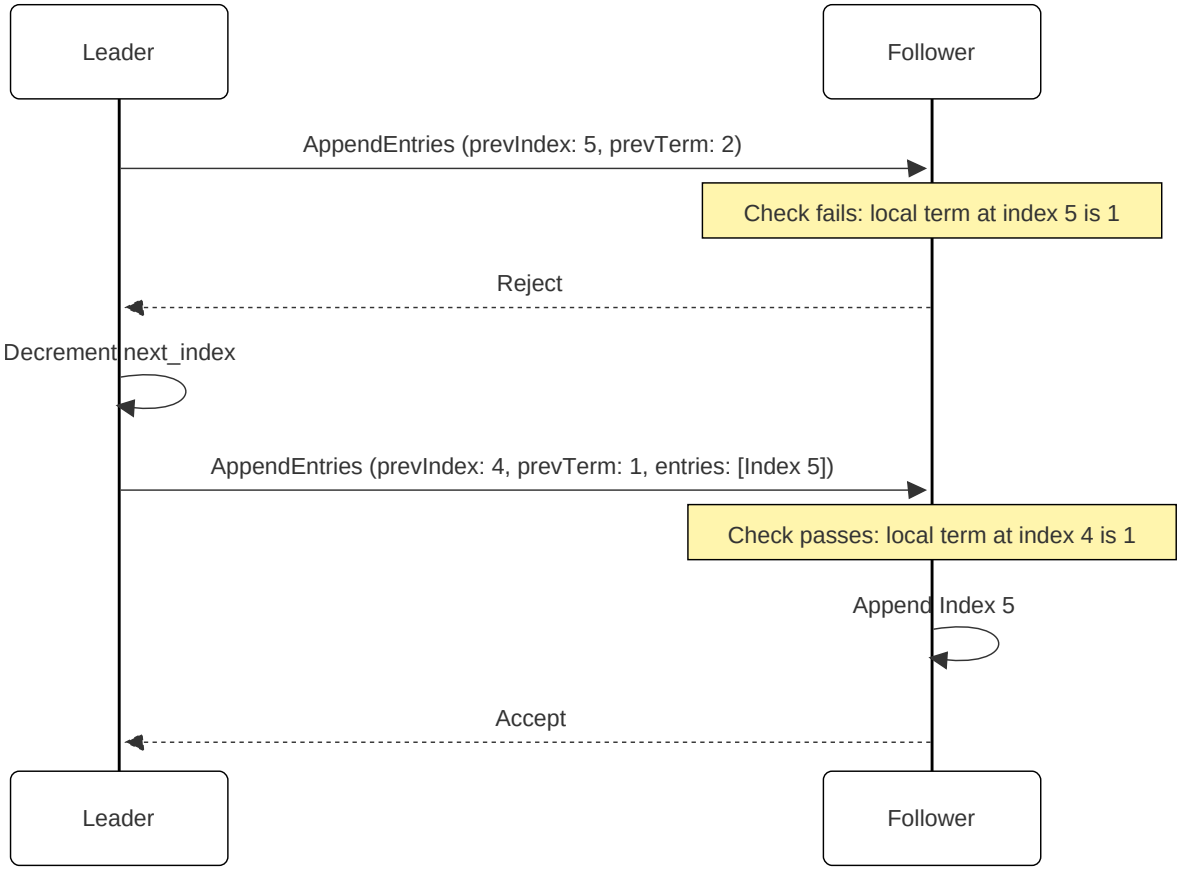

Figure 2.4: Sequence diagram of the Raft log matching phase. When a follower detects a mismatch between its local log and the leader's prev_log_index or prev_log_term, it rejects the AppendEntries request. The leader then decrements the next_index for that follower and retries the RPC with an earlier log entry. This consistency check ensures that the follower synchronizes its state with the leader before any new entries are appended

2.5 Comparative Analysis

The main differences between VSR and Raft are how primaries are selected and the recovery phase.

2.5.1 Primary Selection

VSR's concept of "view" and Raft's concept of "term" are analogous. In VSR the next primary is pre-determined, therefore it may be missing some information. Other nodes must share their state with it to ensure its log is updated. In Raft, only nodes with the most updated state may be elected, as nodes will not vote for peers that have less information than them.

2.5.2 Log Replication

A key difference between log replication in VSR and Raft is that Raft's AppendEntries messages are cumulative. If an entry fails to get replicated, the following AppendEntries messages will resend that entry until it is received. On the other hand, if a PREPARE message is lost, a backup will need to explicitly download the missing state before it can apply newer PREPARE messages.

This is why Raft can use AppendEntries as heartbeats. If entries failed to get replicated, the leader will resend them periodically. If no entries are to be sent, the heartbeats can be empty. This is not possible in

VSR.

2.5.3 Recovery

In Raft there is no node recovery phase. When a node restarts after a crash, it simply reloads its state from disk and rejoins the cluster. VSR nodes on the other hand must download the current state from the primary in order to rejoin, which is more involved.

Using persistent logs that can be reloaded on recovery simplifies the restart process as it makes crashes indistinguishable from slow nodes. This allows Raft to be simpler as it does not need a distinct recovery mechanism.

However, if a node were to lose its state due to disk failure, it could not join the cluster as losing information may cause the safety invariants to break. For instance, losing the information of who the node voted for before crashing may lead it to vote twice in a single term.

On the other hand, in VSR nodes that crash always need to rebuild their state from scratch. Since VSR also handles slow clients, it is trivial to extend the protocol by adding a log. This will cause restarted nodes to skip the recovery protocol. Unlike Raft, the log is entirely optional. If disk failure were to occur, a node could simply rebuild the system's state from scratch.

Chapter 3

Implementation of Raft and VSR

The previous chapter presented VSR and Raft from a theoretical perspective, highlighting their similarities and key differences. To conduct a fair and controlled comparative study, I developed custom implementations of both the Viewstamped Replication (VSR) and Raft consensus algorithms from scratch in C. This chapter details the architecture, design decisions, and data structure of my implementations.

Several implementations exist for Raft, most notably LogCabin[10], the original reference implementation, and etcd/raft[11] used by Kubernetes. On the other hand, VSR has seen few standalone open-source libraries. The most notable one is the one backing TigerBeetle[12], a financial database. These implementations are optimized for specific

use-cases in mind and designed for specific systems. Using existing implementations would not yield a fair comparison, which is why I opted for new implementations.

The source code for both protocols, along with the shared networking and storage layers, is available at:

`https://github.com/cozis/Thesis_Code`

3.1 Architecture & Design

3.1.1 Message-Passing system

Communication between nodes uses a message-based protocol on top of TCP. The message protocol implements a framing mechanism to separate TCP's stream of bytes into separate units. This is done by prefixing a message's length into its header. Receiving nodes can read the fixed-sized header and infer how many incoming bytes are associated to that message.

While UDP was considered for its message-based nature, it was rejected as it does not offer a fragmentation mechanism, and therefore would require it to be implemented from scratch.

3.1.2 Concurrency Model

Implementations are single-threaded, and allow concurrent communication of a node with multiple peers using I/O multiplexing via the `poll()` system call. I/O multiplexing and, more generally, asynchronous I/O allow a program to decouple the number of concurrent I/O operations it can perform from the number of threads.

3.1.3 Timer System

Timeouts are implemented in userspace. The implementations keep track of the next timing deadline and use the timeout argument of `poll()` to wait for events until such deadline.

3.1.4 Storage & Persistence

To guarantee durability across system crashes and restarts, critical state is persisted to non-volatile storage using Write-Ahead Logging (WAL). Operation descriptions are written to a log file. When the writes are flushed to disk using `fsync()`, the process can apply the operation. If a process crashes, it can replay the log, reaching the same state it was in when the crash occurred.

Log entries must be preceded by their length. This allows readers to detect partial writes to the log in cases of crashes during a write. It is essential that operations are logged before operations are applied, not

after. This adds the constraint that disk operation must be performed synchronously.

Data is serialized using a lightweight custom binary format.

3.1.5 State Machine

Both Raft and VSR are designed to replicate arbitrary deterministic state machines. The choice of state machine is therefore orthogonal to the consensus protocol itself, but it must be complex enough that bugs in the replication layer can surface as observable inconsistencies. For this reason, both implementations use a fixed-capacity key-value store as their replicated state machine. The store supports four operations:

- NOOP: no effect
- SET: associates a value with a key
- GET: retrieves the value associated with a key
- DEL: removes an entry

The capacity is fixed at 64 entries, which is sufficient to exercise a meaningful range of state transitions while keeping the implementation simple.

3.1.6 Client Interaction

Clients connect to the current primary and issue requests sequentially: each operation may only have one in-flight request at a time. Requests have IDs that are incremented monotonically, which combines with the client table mechanism described in Chapter 2, ensures requests are not applied twice or out-of-order.

In this implementation, clients generate random operations against the key-value store (SET, GET, and DEL with random keys and values). This ensures a wide range of state machine transitions is exercised during testing.

If a client's connection to the primary is lost, it must discover the new primary. In the current implementation, clients handle this by cycling through the known node addresses until a connection is accepted.

3.1.7 Configuration

The cluster topology is defined statically at startup. Each node receives the list of node addresses in the cluster as a command-line argument. This list is sorted and used to derive a deterministic index for each node, which is referenced throughout the protocol. No dynamic reconfiguration mechanism is implemented: nodes cannot be added or removed while the system is running. This is a deliberate simplification.

3.2 Viewstamped Replication

In this section, we examine the VSR implementation, focusing on its explicit state management for recovery and view changes, and its linear commit mechanism.

3.2.1 Data Structures

The core of the VSR node is the `NodeState` structure. Unlike Raft, which handles node restarts implicitly via persistent storage, VSR requires explicit `recovery_` field to rebuild state from peers. The `status` field is critical as it dictates whether the node is in `NORMAL` operation, performing a `CHANGE_VIEW`, or running the `RECOVERY` protocol.

```
typedef struct {

    // Distinct status enum:
    //   - STATUS_NORMAL
    //   - STATUS_CHANGE_VIEW
    //   - STATUS_RECOVERY
    Status status;

    // Monotonically increasing view number
    uint64_t view_number;

    // Recovery State: VSR nodes must rebuild state
    // from peers after a crash
    uint32_t recovery_votes;
    uint64_t recovery_nonce;
```

```
    Log      recovery_log;

    // View Change State: separate buffers for
    // the transition protocol
    Log      view_change_log;
    int      view_change_commit;

    // The primary log of operations
    Log log;

    // ...
} NodeState;
```

3.2.2 Processing PREPARE_OK messages

During normal operation, the primary waits for a quorum of `PREPARE_OK` messages. The implementation uses a bitmask (`votes`) to ensure each node counts only once. Once a majority is reached, the `commit_index` is advanced. This straightforward counting contrasts with Raft's need to track individual follower indexes. When an operation is committed, a completion response is sent out to the client with the result of the operation. When the primary will send out the next heartbeat, backups will be aware of the new commit number and commit their head of the log.

```
static HandlerResult
process_prepare_ok(NodeState *state, int conn_idx,
 ↪  ByteView msg)
{
    // ... parsing code ...

    int i = message.log_index;

    // Standard check for stale messages
    if (i < state->commit_index)
        return HR_OK;

    LogEntry *entry = &state->log.entries[i];

    add_vote(&entry->votes, message.sender_idx);

    if (reached_quorum(state, entry->votes)) {
        advance_commit_index(state, i + 1, true);
    }
    return HR_OK;
}
```

3.3 Raft

In this section, we present the Raft implementation, highlighting the volatile state maintained by leaders and the Log Matching property that ensures safety without explicit state transfer protocols.

3.3.1 Data Structures

The Raft `NodeState` differs significantly from VSR in that the leader maintains specific progress indices (`next_indices` and `match_indices`) for every peer. This allows the leader to manage the "sliding window" of replication for each follower independently.

```
typedef struct {

    Role role; // FOLLOWER, CANDIDATE, LEADER

    // The following fields are volatile state
    // used by leaders. They are reinitialized
    // after the election.

    // Index of the next log entry to send to
    // that server
    int next_indices[NODE_LIMIT];

    // Index of highest log entry known to be
    // replicated on server
    int match_indices[NODE_LIMIT];

    // ...
} NodeState;
```

Crucial to Raft's design is the `AppendEntriesMessage`, which carries the `prev_log_index` and `prev_log_term`. These fields act as a "guard", allowing the follower to verify the continuity of the log before accepting new entries.

```
typedef struct {
    MessageHeader base;
    uint64_t term;
    int      leader_idx;

    // The "Consistency Check" fields:
    int      prev_log_index;
    uint64_t prev_log_term;

    int      leader_commit;
    // ...
} AppendEntriesMessage;
```

3.3.2 Log Replication & Log Matching

The safety of Raft relies on the Log Matching Property. The following snippet demonstrates the "Induction Step" of this property: a follower only accepts new entries if it finds a log entry at `prev_log_index` with the matching `prev_log_term`. If this check fails, the message is rejected, prompting the leader to decrement its `next_index` for this follower.

```
// Perform the log consistency check
if (prev_log_index >= 0) {

    // 1. Check if the log is long enough
    if (prev_log_index >= wal_entry_count(&state->wal)) {
        // We don't have the entry at prev_log_index
        return send_appended_response(state, conn_idx,
        ↪  false, -1);
    }

    // 2. Check if the term matches (The Log Matching
    ↪  Property)
    if (wal_peek_entry(&state->wal, prev_log_index)->term
    ↪  != prev_log_term) {
        // Conflicting entry at prev_log_index
        return send_appended_response(state, conn_idx,
        ↪  false, -1);
    }
}
```

Chapter 4

Validation

Validating consensus algorithms is notoriously difficult. Standard unit testing is insufficient because bugs often arise from complex interleavings of network events and failures that are hard to reproduce. To address this, this study adopts a validation methodology based on Deterministic Simulation Testing (DST), invariant checking, and fault injection.

4.1 Deterministic Simulation

Instead of running nodes as separate processes communicating over real network loopback, a custom simulation framework named Quakey was developed which offers:

- Single-Threaded Execution: The entire cluster runs in a single

process

- Virtual Time: Time is mocked
- Deterministic Scheduling: Nodes are effectively scheduled in userspace, allowing interleavings to be completely deterministic

This ensures runs of the same cluster that are seeded in the same way will produce the same results.

4.2 Fault Injection

To verify the resilience of Raft and VSR, the simulation environment injects failures:

1. Packet Reordering: The simulation framework randomly delays packets. Generally speaking, it would be possible to also test packet loss, but that is not necessary as the implementations leverage the OS TCP implementation.
2. Node Crashes: Nodes are forcefully crashed at random times.
3. Disk Failures: Although not necessary for these specific implementations, the framework can arbitrarily introduce partial disk failures in the form of bad writes or bitrot.

4.3 Safety Verification

Merely checking that the system does not crash is not enough. The system must maintain safety properties at all times.

The simulating logic is capable of inspecting all node states simultaneously at any given instant of time. This allows it to check that per-node and inter-node invariants of the protocol are never broken.

The strongest safety check is the shadow log. The simulator holds a copy of the primary's log, which is referred to as "shadow log". Whenever a view changes or a recovery happens, the simulation ensures that the previously committed entries are not lost. The shadow log check runs after each node scheduling. It checks that all previously added entries were not lost or corrupted, and then adds the newly committed entries to it.

Both implementations were subjected to over 200,000 simulated seconds (2 days and 7 hours) of operation each, with fault injection active throughout. All invariants were checked after each node scheduling step, and no violations were observed across the entire duration of the simulation.

Chapter 5

Conclusion

This thesis compared Viewstamped Replication and Raft, two replication algorithms that are architecturally similar but designed decades apart with different design constraints in mind.

Chapter 2 presents the two protocols theoretically. Both are leader-based state machine replication algorithms. In VSR, leaders are selected deterministically in round-robin fashion, while in Raft they are elected randomly. Since primary selection in VSR is predetermined, it is possible for it to need a state update before it can act as primary, creating the need for a state transfer. In Raft, only nodes that are up-to-date can be elected. This state transfer mechanism adds complexity to VSR but also allows it to restore state in case of storage faults, which would be fatal for a Raft cluster.

Chapter 3 presents two implementations of the protocols written in C.

Both implementations use a shared networking, storage, and state machine code. This simplifies the comparison of the protocols themselves rather than different implementation details. The specific state that is replicated is orthogonal to the protocol design, so a fixed-capacity key-value store was chosen as it is complex enough to cause implementation errors to surface as incoherent key values.

Chapter 4 presents the testing and validation strategies employed to ensure the algorithm's correctness. A simulation system was used to run all nodes of the system deterministically in a single process. All networking, storage and scheduling is performed in userspace and in-memory. Arbitrary faults are injected into the simulation to ensure the implementations are robust. Injected faults include network partitions, delays, I/O errors, and crashes. Both implementations were validated over 200,000 simulated seconds each, during which all safety invariants were continuously verified without a single violation. Since simulation time is virtual, dead periods where nodes are inactive can be fast-forwarded, allowing simulation time to advance faster than real-time.

Bibliography

[1] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming.* Springer, 2011.

[2] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column)*, vol. 32, no. 4, pp. 51–58, 2001.

[3] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[4] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, 1988.

[5] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[6] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Transactions on*

Database Systems (TODS), vol. 4, no. 2, pp. 180–209, 1979.

[7] D. K. Gifford, "Weighted voting for replicated data," in *Proceedings of the seventh ACM symposium on Operating systems principles*, 1979.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[9] B. Liskov and J. Cowling, "Viewstamped replication revisited," *MIT-CSAIL-TR-2012-021*, 2012.

[10] D. Ongaro, "LogCabin." `https://github.com/logcabin/logcabin`. Accessed: 2025.

[11] etcd Authors, "etcd/raft." `https://github.com/etcd-io/raft`. Accessed: 2025.

[12] TigerBeetle, Inc., "TigerBeetle." `https://github.com/tigerbeetle/tigerbeetle`. Accessed: 2025.