Bear-bones Log Replication Project Report

Distributed Systems Course 2360351

Student Name: George Nasser

Student ID: 324830587

Winter 2025/26

1 Introduction

1.1 Problem Statement

The system must ensure that:

- All replicas agree on the order of operations (total ordering)
- The system continues to function when up to $f < n/2$ servers fail
- Failed servers can recover and catch up to the current state
- Clients receive consistent responses regardless of which server they contact

1.2 Solution Overview

I implemented a **State Machine Replication** approach using the **Paxos consensus protocol**. The key insight is that if all replicas:

1. Start in the same initial state
2. Execute the same commands in the same order
3. Have deterministic state transitions

Then all replicas will end up in the same final state.

Paxos provides the mechanism to ensure all replicas agree on the order of commands, even in the presence of failures.

2 Implementation Journey

This section describes how the project was developed step-by-step, following the recommended stages from the assignment.

2.1 Stage 0: Learning & Setup

Before implementation, I studied the necessary technologies:

1. **Go Language**: Completed the Go Tour to learn Go Language and its features.
2. **gRPC**: Followed the gRPC Go Quickstart to understand how to define services using Protocol Buffers and implement RPC communication between servers.
3. **Paxos Protocol**: Studied the lecture slides to understand the three-phase consensus protocol (Prepare, Accept, Commit) and why it guarantees safety even with failures.

2.2 Stage 1: Basic Paxos Implementation

2.2.1 Protocol Messages (paxos.proto)

I defined the gRPC messages in Protocol Buffers:

```
service Paxos {
    rpc Prepare(PrepareRequest) returns (PromiseResponse);
    rpc Accept(AcceptRequest) returns (AcceptedResponse);
    rpc Commit(CommitRequest) returns (CommitResponse);
}
```

Each message serves a specific purpose in the protocol:

- `PrepareRequest`: Proposer asks acceptors to promise not to accept older proposals
- `PromiseResponse`: Acceptor returns any previously accepted value
- `AcceptRequest`: Proposer asks acceptors to accept a value
- `CommitRequest`: Tells all servers the final decided value

2.2.2 Acceptor Implementation (acceptor.go)

The acceptor maintains state for each consensus instance:

```go
type AcceptorInstance struct {
    lastRound     []int64
    lastGoodRound []int64
    v_i           int64
    decided       bool
    decidedValue  int64
}
```

Why these variables?

- `lastRound`: Ensures we only respond to proposals with higher round numbers - prevents old proposals from succeeding
- `lastGoodRound` and `v_i`: Allows the proposer to learn about any previously accepted values
- `decided`: Prevents re-deciding an instance

2.2.3 Proposer Implementation (proposer.go)

The proposer coordinates the three-phase protocol:

Phase 1 - Prepare:

1. Generate a unique round number $[sequence, server\_id]$
2. Send `PrepareRequest` to all acceptors
3. Wait for majority of promises
4. If any acceptor already accepted a value, use the value from the highest round

Phase 2 - Accept:

1. Send `AcceptRequest` with the chosen value
2. Wait for majority of accepts

Phase 3 - Commit:

1. Broadcast `CommitRequest` to all servers
2. Servers apply the command to their state machine

2.3 Stage 2: Failure Detection & Membership

2.3.1 Why etcd?

The assignment required using etcd for coordination. etcd provides:

- **Key-value storage**: Store server addresses under `members/{id}`
- **Leases with TTL**: Automatic cleanup when servers crash
- **Watch mechanism**: Real-time notifications when membership changes
- **Consistency**: etcd itself uses Raft, guaranteeing consistent membership views

2.3.2 Lease Based Failure Detection

Each server:

1. Obtains a lease from etcd with 5-second TTL
2. Registers itself: `PUT members/{id} = address`
3. Sends periodic heartbeats to renew the lease

If a server crashes:

1. It stops sending heartbeats
2. After 5 seconds, the lease expires
3. etcd automatically deletes the key
4. Other servers are notified via Watch

This provides **automatic failure detection**

2.4 Stage 3: Multi-Paxos Extension

2.4.1 Why Multi-Paxos?

Basic Paxos decides on a single value. For a replicated log, we need to decide on a sequence of commands. Multi-Paxos runs a separate Paxos instance for each log entry:

- Instance 0 decides the first command
- Instance 1 decides the second command
- Instance n decides the $(n + 1)$th command

2.4.2 Implementation

I modified the acceptor to maintain a map of instances:

```
type Acceptor struct {
    instance map[int64]*AcceptorInstance  // One per log slot
    ...
}
```

Each Paxos message now includes an `instance_id` field to specify which log slot is being decided.

2.5 Stage 4: Replicated Log & Scooter Rental App

2.5.1 Scooter Rental Application

The state machine supports three commands:

Command	Effect	Constraints
CREATE	Add new scooter	Scooter must not exist
RESERVE	Mark scooter unavailable	Scooter must be available
RELEASE	Mark scooter available, add distance	Scooter must be reserved

2.5.2 Replicated Log Structure

```
type ReplicatedLog struct {
    entries     map[int64]*LogEntry
    nextIndex   int64                   // Next available slot
    commitIndex int64                   // Highest committed entry
    storedIndex int64                   // Explained Later
}
```

2.6 Stage 5: Log Replication

The key insight is that **commands are replicated via Paxos, not just the values**. During the Commit phase, the actual command bytes are sent to all servers:

```
message CommitRequest {
    int64 value = 1;
    int64 instance_id = 2;
    bytes command = 3;  // The actual command to replicate
}
```

When an acceptor receives a Commit:

1. Append the command to its local log
2. Apply the command to its state machine

This ensures all servers have identical logs and identical state.

2.7 Stage 6: Node Recovery

2.7.1 The Recovery Problem

When a server crashes and restarts (or a new server joins), it has an empty log and state. It needs to:

1. Discover what commands were decided before it has joind (again)
2. Apply those commands to catch up to the current state

2.7.2 Recovery Protocol

I implemented a gRPC service for log recovery:

```
service LogRecovery {
    rpc GetLog(GetLogRequest) returns (GetLogResponse);
}
```

The recovering server:

1. Sends its current `nextIndex` to another server
2. Receives all log entries from that index onward
3. Applies each entry to its log and state machine

2.8 Stage 7: Snapshots & Log Compaction

2.8.1 The Problem with Unbounded Logs

Without log compaction, the log grows forever:

- Memory usage increases indefinitely
- Recovery takes longer (must replay entire history)
- Old entries are never garbage collected

2.8.2 Solution: Snapshots

A snapshot captures the state machine at a point in time:

```
func (sm *ScooterStateMachine) TakeSnapshot(index int64) error {
    data, err := json.Marshal(sm.scooters)
    sm.snapshotData = data
    sm.snapshotIndex = index
}
```

2.8.3 Snapshot-Aware Recovery

If a recovering server is too far behind:

1. Send the snapshot instead of old log entries
2. The server loads the snapshot
3. Then replays entries after the snapshot index

2.9 Stage 8: Bug Fixes & Optimizations

Several bugs were discovered and fixed during testing:

2.9.1 Bug 1: Proposer Not Voting for Itself

Problem: The proposer counted itself in the majority calculation but never actually called its local acceptor.

Consequence: With 5 servers, we need 3 for majority. If the proposer didn't vote, we effectively needed 3 out of 4 remote servers (75% instead of 60%).

Fix: Added `localAcceptor` field and explicit calls:

```go
localPromise, _ := p.localAcceptor.Prepare(ctx, req)
if localPromise.Ack {
    promises = append(promises, localPromise)
}
```

2.9.2 Bug 2: Double State Application

Problem: The API handler called `Apply()` after `Propose()`, but the Commit handler also called `Apply()`.

Consequence: State was applied twice on the proposing server, causing incorrect distance calculations.

Fix: Removed `Apply()` from handlers; let only Commit handle state application.

2.9.3 Bug 3: Race Condition in GetNextIndex

Problem: `GetNextIndex()` returned the index without incrementing it.

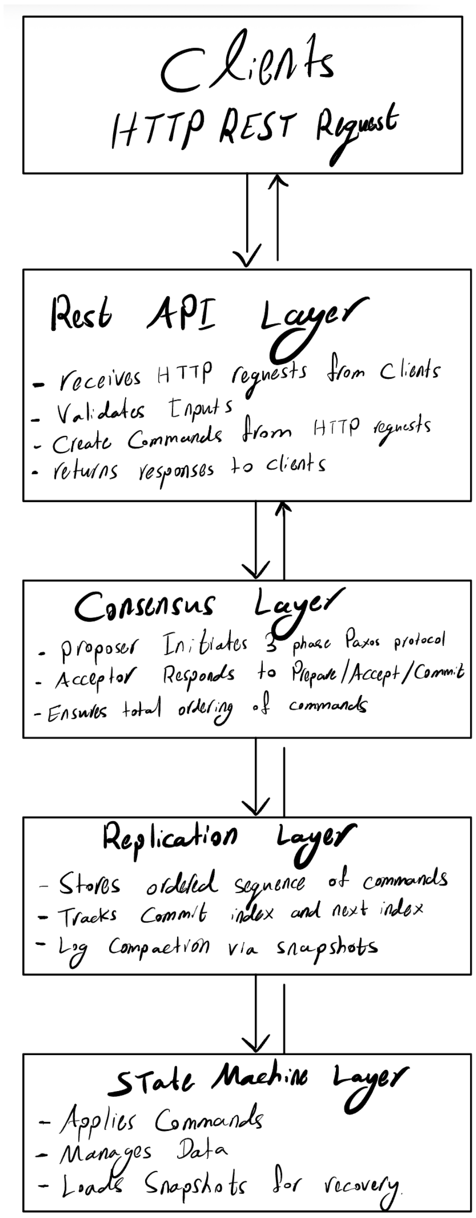
Consequence: Concurrent requests got the same index, trying to decide the same Paxos instance with different values.

Fix: Made `GetNextIndex()` atomic:

```go
func (log *ReplicatedLog) GetNextIndex() int64 {
    log.mutex.Lock()
    defer log.mutex.Unlock()
    index := log.nextIndex
    log.nextIndex++
    return index
}
```

3 System Architecture

3.1 High-Level Architecture


Clients
HTTP REST Request
Rest API Layer
- receives HTTP requests from Clients
- Validates Inputs
- Create Commands from HTTP requests
- returns responses to clients
Consensus Layer
- proposer Initiates 3 phase Paxos protocol
- Acceptor Responds to Prepare/Accept/Commit
- Ensures total ordering of commands
Replication Layer
- Stores ordered sequence of commands
- Tracks Commit index and next index
- Log Compaction via snapshots
State Machine Layer
- Applies Commands
- Manages Data
- Loads Snapshots for recovery.

3.2 Component Interactions

3.2.1 Write Path (e.g., Reserve Scooter)

When a client sends `POST /scooters/123/reservations`:

1. **API Handler** receives request, validates scooter exists and is available
2. **API Handler** creates a `RESERVE` command and gets next log index
3. **Proposer** starts Paxos Phase 1 (Prepare) for that index
4. **Acceptors** respond with promises
5. **Proposer** starts Paxos Phase 2 (Accept) if majority promised
6. **Acceptors** respond with accepts
7. **Proposer** broadcasts Commit to all servers
8. **All Acceptors** append to log and apply to state machine
9. **API Handler** returns success to client

3.2.2 Read Path (e.g., Get Scooter)

When a client sends `GET /scooters/123`:

1. **API Handler** receives request
2. **API Handler** reads directly from local state machine
3. **API Handler** returns scooter data to client

Reads do not go through Paxos because:

- They don't modify state
- Sequential consistency is acceptable for reads
- It is faster

3.3 Network Communication

The system uses two communication protocols:

Protocol	**Between**	**Purpose**	**Port**
HTTP/REST	Client ↔ Server	Client operations	8081-8085
gRPC	Server ↔ Server	Paxos & Recovery	50051
gRPC	Server ↔ etcd	Membership	2379

Different protocols

- **HTTP/REST for clients**:
- **gRPC for servers**

4 Consistency Guarantees

The assignment requires implementing at least one operation from each consistency category. Our implementation provides:

Consistency Level	Operations	Implementation
Linearizable Write	CREATE, RESERVE, RELEASE	Via Paxos consensus
Sequentially Consistent Read	GET (default)	Local state machine read
Linearizable Read	GET with ?linearizable=true	NOOP via Paxos then read

4.1 Linearizable Writes

All write operations (CREATE, RESERVE, RELEASE) are **linearizable**.

Implementation in handlers.go:

```go
func (api *API) CreateScooter(context *gin.Context) {
    cmd := statemachine.ScooterCommand{
        CommandType: statemachine.Create,
        ScooterID: scooterID,
    }
    cmdBytes, _ := json.Marshal(cmd)
    index := api.log.GetNextIndex()
    _, err := api.proposer.Propose(index, index, cmdBytes)
}
```

Why this is linearizable:

1. Each write gets a unique log index via atomic `GetNextIndex()`
2. `Propose()` runs full 3-phase Paxos (Prepare → Accept → Commit)
3. The call blocks until majority accepts and Commit is broadcast
4. All servers apply commands in log index order
5. Therefore, the write appears atomic at some point during the call

4.2 Sequentially Consistent Reads

By default, read operations return the local state machine value.

Implementation in handlers.go:

```go
func (api *API) GetScooter(context *gin.Context) {
    scooter, exists := api.stateMachine.GetScooter(context.Param(
       "id"))
    if !exists {
        context.JSON(http.StatusNotFound, gin.H{"error": "Scooter
           ␣not␣found"})
        return
    }
    context.JSON(http.StatusOK, scooter)
}
```

Why this is sequentially consistent:

- The read returns all writes that were committed on this server
- Writes are applied in log order (total order from Paxos)
- A lagging server may not have all commits yet, but sees a consistent prefix
- This satisfies sequential consistency: reads see writes in program order

4.3 Linearizable Reads (Optional - idea from Claude)

For stronger consistency, clients can request linearizable reads via query parameter.

Implementation in handlers.go:

```go
func (api *API) GetScooter(context *gin.Context) {
    // Check if linearizable read is requested
    if context.Query("linearizable") == "true" {
        // Run a NOOP through Paxos to ensure we see all prior commits
        cmd := statemachine.ScooterCommand{
            CommandType: statemachine.Noop,
        }
        cmdBytes, _ := json.Marshal(cmd)
        index := api.log.GetNextIndex()
        _, err := api.proposer.Propose(index, index, cmdBytes)
        if err != nil {
            context.JSON(http.StatusInternalServerError,
                gin.H{"error": "Failed␣to␣ensure␣linearizability"})
            return
        }
    }
    scooter, exists := api.stateMachine.GetScooter(context.Param("id"))
    context.JSON(http.StatusOK, scooter)
}
```

Implementation in scooter.go (NOOP command):

```go
const (
    Create  = "CREATE"
    Reserve = "RESERVE"
    Release = "RELEASE"
    Noop    = "NOOP"   // For linearizable reads
)

func (sm *ScooterStateMachine) Apply(commandBytes []byte) error {
    // ...
    case Noop:
        // No-op: does nothing, used for linearizable reads
        // ...
}
```

Why this is linearizable:

1. Before reading, we propose a NOOP command through Paxos
2. This NOOP must wait for any in-flight writes to complete
3. After NOOP commits, our server has all previously committed values
4. The subsequent read returns the most recent committed state
5. The read appears to happen at the moment the NOOP commits

Usage:

```
# Default sequentially consistent read
GET /scooters/123

# Linearizable read (slower but strongly consistent)
GET /scooters/123?linearizable=true
```

5 Failure Handling

5.1 Server Crash During Paxos

If a server crashes during a Paxos round:

Proposer crashes after Prepare, before Accept:

- No value was accepted
- Another proposer can start a new round
- The new round will succeed normally

Proposer crashes after some Accepts:

- Some acceptors have accepted the value
- Another proposer starts a new round
- The new proposer will discover the accepted value via Prepare promises
- The new proposer will use that value, ensuring consistency

Acceptor crashes:

- If majority still available, consensus continues
- If less than majority, consensus blocks until recovery

5.2 Network Partitions

If the network splits:

- The split with majority can continue operation
- The minority partition cannot reach consensus (not enough acceptors)
- When partition heals, minority servers recover via log recovery

5.3 Recovery After Crash

When a server restarts:

1. Re-registers with etcd (new lease)
2. Calls `Recover()` to catch up
3. Contacts other servers for missing log entries
4. Applies entries to local state
5. Resumes normal operation

6 Docker Deployment

6.1 Container Architecture

The system runs as Docker containers orchestrated by Docker Compose:

Container	Image	Ports	Purpose
scooter-server-1	scooter-server:0.3	8081	Server replica 1
scooter-server-2	scooter-server:0.3	8082	Server replica 2
scooter-server-3	scooter-server:0.3	8083	Server replica 3
scooter-server-4	scooter-server:0.3	8084	Server replica 4
scooter-server-5	scooter-server:0.3	8085	Server replica 5
etcd	quay.io/coreos/etcd:v3.5.9	2379	Coordination service
spa	scooter-spa:0.1	4300	Web frontend

6.2 Network Configuration

All containers share a bridge network (`scooter-net`):

- Containers can reach each other by name (e.g., `scooter-server-1`)
- Internal gRPC uses port 50051
- External HTTP uses ports 8081-8085

6.3 Running the System

```
# Build the server image
docker build -f Dockerfile.server -t scooter-server:0.3 .

# Start all containers
cd src/docker
docker-compose up -d

# View logs
docker-compose logs -f scooter-server-1
```

7 Design Decisions Summary

1. **Round Numbers as Tuples** $[sequence, server\_id]$
   - Ensures global uniqueness without coordination
   - Enables deterministic comparison
   - Server ID breaks ties
2. **Local Acceptor Optimization**
   - Proposer calls local acceptor directly (no gRPC)
   - Reduces latency by avoiding network round-trip
   - Still counts toward quorum
3. **Synchronous Local Commit**
   - Proposer waits for local commit before returning to client
   - Ensures proposing server has latest state
   - Simplifies read-after-write consistency
4. **Atomic Index Allocation**
   - `GetNextIndex()` atomically returns and increments
   - Prevents race conditions in concurrent requests
   - Each request gets a unique log slot
5. **Command Replication in Commit**
   - Commands are sent with Commit, not Accept
   - Reduces message size during consensus
   - Commands only replicated after decision
6. **etcd for Membership Only**
   - Per assignment: etcd for coordination, not for ordering transactions
   - Paxos handles command ordering
   - etcd handles failure detection

8 Known Limitations

1. **In-Memory Storage**
    - State is not persisted to disk
    - Server restart requires full recovery from other servers
    - Acceptable per assignment (allows in-memory storage)
2. **No Leader Optimization**
    - Every write runs full 3-phase Paxos
    - Multi-Paxos optimization (stable leader skips Phase 1) not implemented
    - Higher latency but simpler implementation
3. **Fixed Cluster Size**
    - 5 servers configured statically
    - No dynamic cluster membership changes
    - Would require reconfiguration protocol
4. **No Read Repair**
    - Reads from local state only
    - Stale reads possible on lagging servers
    - Could add read-repair or quorum reads

A File Locations Reference

Feature	**File**
Paxos Proposer	`src/server/paxos/proposer.go`
Paxos Acceptor	`src/server/paxos/acceptor.go`
Replicated Log	`src/server/log/replicated_log.go`
State Machine	`src/server/statemachine/scooter.go`
REST API	`src/server/api/handlers.go`
Membership/etcd	`src/server/membership/membership.go`
Recovery	`src/server/recovery/recovery.go`
Proto Definitions	`src/server/proto/paxos.proto`
Main Entry	`src/server/main.go`
Docker Compose	`src/docker/docker-compose.yml`

Table 1: Complete File Location Reference