DAD Project Report 2025/2026

João Pereira - ist1112175 Martim Pereira - ist1112272 João Tiago - ist199986

1. Introduction

This report describes the final implementation of the dida-meetings system. We implemented Step 2 (Multi-Paxos with concurrent Phase 2 proposals), Step 3 (Vertical Paxos II for reconfigurable leadership), and Step 4 (fast topic operations bypassing consensus). Key design choices include thread pools for parallel command proposals, a master console coordinating leader changes, and a pending topic list for low-latency updates.

2. The Problem

The dida-meetings system requires reliable replication and dynamic reconfiguration. In our implementation:

- **Step 2 – Multi-Paxos:** Commands are proposed concurrently to improve throughput.
- **Step 3 – Vertical Paxos II:** The console manages leadership and configuration changes, ensuring state transfer between leaders.
- **Step 4 – Fast Topic Operations:** Lightweight topic commands are executed locally to reduce latency while maintaining consistency for other operations.

3. Implementation

3.1. Changes to the original code

For a better understanding, the original code has been divided into smaller classes to encapsulate logic and follow the Paxos nomenclature. We now have three distinct server functions: ***Proposer***, ***Acceptor***, and ***Learner***, each with its own logic, as described in the **"Paxos Made Simple"** paper. This logic was originally in the ***MainLoop*** class and on the ***PaxosServiceImp*** class.

For auxiliary purposes, classes of reply and result were created for each phase of the Paxos algorithm. These classes store the necessary information for the communication on the algorithm; They store:

- `maxballot`: the largest ballot that the acceptor has promised to work with.
- `Map<Integer, InstanceInfo>`: stores the pair of `valballot, value` that were accepted on other instances.

3.2. Step 2: Multi-Paxos Implementation

3.2.1 Proposer

The proposer was extended to support multi-instance consensus:

- **Leader Phase 1 Tracking:** Ensures Phase 1 executes only once per leadership period, avoiding redundant ballot preparations.
  We maintain a boolean on the Proposer to track whether Phase 1 has been completed for the current ballot. This variable is set to true in two cases: first, if `completedBallot < 0`, meaning there was no previous activity and Phase 1 can be assumed done; second, if `completedBallot > 0`, after performing a proper Phase 1 and succeeding, the variable is set to true to indicate successful completion. In all other cases, it remains false until Phase 1 finishes successfully. The variable is set to false if the replica is not the leader for the current ballot or if a phase one didn't succeed
- **Gap Handling:** Undecided log entries are filled with **NOOPs** to maintain continuity.

```
1  if (entry.isDecided()) {
2      int finalI = i;
3      futures.add(executor.submit(() ->
           startPhaseTwo(finalI, ballot, entry.
           getCommand_id())));
4      continue;
5  }
6  if (!orderedMap.containsKey(i)) {
7      int finalI = i;
8      futures.add(executor.submit(() ->
           startPhaseTwo(finalI, ballot, NOOP)))
           ;
9  } else {
10     InstanceInfo info = orderedMap.get(i);
```

```
    int valueToPropose = (info.valBallot > -1) ? info.value : NOOP;
    int finalI = I;
    futures.add(executor.submit(() -> startPhaseTwo(finalI, ballot, valueToPropose)));
}
```

Listing 1: Log entries check

- **Concurrent Phase 2 Proposals:** Uses a thread pool to submit multiple Phase 2 proposals in parallel.
- **Internal Data Structures:** Inside the RequestHistory class, there are three Hashtables: one for pending requests, one for in-progress requests (placed there only by the leader when starting an instance for it), and one for processed requests.

  A pending request will be marked as in progress by the leader when running a phase 2 for it, and by learners before executing the command after being decided. This way, before executing a command, we can pass it from in progress to processed, keeping track of RequestRecord's whose command has been executed.

  We also rollback a request from in progress to pending in the leader replica whenever a phase 2 isn't accepted, allowing this replica to, in the future, after becoming the leader again, attempt to propose it again.

3.2.2 Acceptor

The Acceptor handles Phase 1 and Phase 2 requests:

- Atomically updates the current ballot to maintain ordering.
- Returns previously accepted values to support leader state transfer.

```
// On Acceptor
for (Map.Entry<Integer, PaxosInstance> e : this.serverState.getPaxos_log().getLog().entrySet()) {
    int instance = e.getKey();
    PaxosInstance entry = e.getValue();
    synchronized (entry) {
        if (accepted) {
            entry.setRead_ballot(ballot);
        }
    }
    instanceMap.put(instance, new InstanceInfo(entry.getCommand_id(), entry.getWrite_ballot()));
}


// On Phase One Processor
for (DidaMeetingsPaxos.InstanceState instanceState:last_response.getInstancesList()) {
    int instance = instanceState.getInstance();
    int val = instanceState.getValue();
    int valBallot = instanceState.getValballot();
// If we haven't seen this instance yet, or the valballot is higher, update it
    if (!instanceMap.containsKey(instance) || valBallot > instanceMap.get(instance).valBallot) {
        instanceMap.put(instance, new InstanceInfo(val, valBallot));
    }
}
```

- Updates the Paxos log with accepted commands or NOOPs.

3.2.3 MainLoop

The MainLoop ensures ordered execution of client commands:

- Monitors incoming client requests continuously.
- Executes commands only after consensus is reached.
- Triggers the proposer if the replica is the current leader.
- Maintains sequential execution by tracking pending and decided entries.

3.3. Step 3: Vertical Paxos II Implementation

3.3.1 Motivation

Multi-Paxos assumes static replicas. Vertical Paxos II adds:

- Master-driven configuration management.
- Safe state transfer between leaders.
- Single active configuration at a time.

3.3.2 Design

3.3.2.1 Console as Master:

- Tracks the latest completed ballot.
- Issues *newBallot* commands for leadership changes.
- Activates new configurations only after prior decisions are fully synchronized.

3.3.2.2 Configuration Manager:

Maintains schedules of roles for acceptors, learners, and leaders.

3.3.2.3 Leader State Transfer:

- When a new ballot is initiated, all servers become temporarily inactive.
- The new leader performs a catch-up phase:
  - Runs Phase 1 with acceptors from the previous configuration.

```
List<Integer> prevConfigAcceptors =
    serverState.getScheduler().acceptors
    (completed_ballot);
PhaseOneProcessor phase_one_processor =
    new PhaseOneProcessor(serverState.
    getScheduler());
ArrayList<PhaseOneReply> responses = new
     ArrayList<>();
Collector<PhaseOneReply> collector = new
     Collector<>(responses,
    prevConfigAcceptors.size(),
    phase_one_processor);
PhaseOneRequest request =
    PhaseOneRequest.newBuilder()
    .setInstance(0)
    .setRequestballot(ballot)
    .build();
for (int acceptor : prevConfigAcceptors)
     {
    serverState.getAsync_stubs()[
        acceptor].phaseone(request, new
        CollectorStreamObserver<>(
        collector));
}
```

  - Proposes Phase 2 for undecided entries to synchronize the log.

```
PaxosInstance entry = serverState.
    getPaxos_log().testAndSetEntry(
    entry_number, ballot);
    if (entry.getCommand_id() == 0) {
    final int finalEntryNumber =
        entry_number;
    final int finalBallot = ballot;
    final int finalRequestId =
        request_record.getId();
    executor.submit(() -> proposer.
        startPhaseTwo(finalEntryNumber,
        finalBallot, finalRequestId));
}
```

  - Tracks completion of all Phase 2 proposals before signaling the master.
- Once all catch-up Phase2s complete, the master sends an *activate* message.
- Only after receiving this activation, the leader can process Phase 2 proposals for new client requests.

This guarantees that no client commands are executed until the system is fully synchronized, preserving correctness after leader changes.

3.4. Step 4: Fast Topic Operation

Fast topic commands bypass Paxos to reduce latency for non-critical updates:

- Executed locally on each replica without consensus.
- Preserves total order for critical commands (OPEN, ADD, CLOSE).

Implementation details:

- Maintains a `List<PendingTopicOperation>`.
- On a new Topic operation, checks if the meeting exists and client is valid.
  - If valid, executes immediately.
  - If not, stores it in the pending list.
- Whenever an OPEN or ADD operation occurs, the pending list is checked for matching operations, which are then executed and removed.
- Topic operations with higher IDs prevail in conflicts.

This design balances performance and consistency: Paxos is applied only for critical commands, while fast topic operations execute independently for low latency.

4. Architecture

The system consists of:

1. **Client:** Issues requests for meetings and participants.
2. **Server:** Runs Paxos roles (Proposer, Acceptor, Learner) and executes commands.
3. **Console:** Master for configuration and leadership management.
4. **Configuration Manager:** Maintains role schedules.

Each server node integrates a **MainLoop** that:

- Coordinates proposals by invoking the proposer when the server is the leader.
- Tracks pending and decided log entries to enforce sequential execution.
- Learns decisions from Phase 2 messages and executes commands in order.

The Console interacts with replicas through control messages, enabling dynamic reconfiguration without requiring system restart.

4.1. Design Decisions

Key design decisions include:

- **Multi-Paxos for Efficiency:** Reduced repeated Phase 1 executions by tracking leadership state and gaps, supporting concurrent Phase 2 proposals.
- **Console-as-Master:** Simplifies coordination for dynamic configuration and ensures a single active configuration at a time.
- **Fast Topic Operations:** Optimizes low-latency commands while preserving total order for critical operations.
- **Thread Pools for Parallelism:** Proposer uses a thread pool for concurrent Phase 2 proposals, trading simplicity for throughput.

4.2. Advantages

- **Throughput:** Multi-Paxos and concurrent Phase 2 proposals improve command processing rates.
- **Safe Reconfiguration:** Vertical Paxos II semantics ensure leader changes and configuration updates occur without violating consistency.
- **Low-Latency Operations:** Fast topic operations execute locally, reducing latency for non-critical updates.
- **Modular Architecture:** Clearly separated roles and responsibilities facilitate future extensions and easier maintenance.

4.3. Limitations

- **Thread Pool Overhead:** Smaller deployments may experience unnecessary context switching or resource usage due to parallel Phase 2 proposals.

5. Conclusion

We presented a Multi-Paxos replication system enhanced with Vertical Paxos II for dynamic reconfiguration and fast topic operations. The system achieves efficient consensus, supports leadership changes, and accelerates lightweight operations without compromising critical consistency guarantees. Future improvements may focus on adaptive concurrency control and improved fault tolerance.

References

[1] L. Lamport. Paxos made simple. 2001.

[2] L. Z. Lamport, Dahlia Malkhi. Vertical paxos and primary-backup replication. *Microsoft Research*, 2019.

[3] L. Rodrigues. Dida meetings. 2025.