CSE550 Final Project Writeup

Paxos State Machine Design
and Implementation

Liangyu Zhao
liangyu@cs.washington.edu

December 14th, 2021

Contents

1 Introduction

This project is an implementation of Paxos state machine from TCP networking to Paxos protocols. Besides standard Paxos algorithm, the implementation adopts two efficiency improvements described in the video by Diego Ongaro: (1) electing leader as the sole proposer, and (2) skipping `PrepareRequest`s once the initial one is accepted. Besides efficiency, the implementation can also handle temporary server failure to: (1) continue processing client requests when minority of servers fail, and (2) recover failed server to the latest state.

2 CSE 452 DsLabs

The design of the framework including the layout of `Node`, `Application`, `Message`, and `Timeout` is borrowed from the DsLabs by Ellis Michael. While the framework of Michael gives author ideas how to layout different elements, the implementation by Michael was only intended to run distributed systems lab assignments on one machine. Thus, the actual implementation to run a Paxos system truly distributed is done by author himself in this project.

Many of the designs of Paxos like batched commands and garbage collection are from the DsLabs assignments done by author when he was taking the course CSE 452 at the University of Washington. The previous implementation is modified to work in a truly distributed environment. In addition, ideas like "up-to-date state" and "`Recover` message" are newly designed and implemented in this project.

3 Design

3.1 Assumptions

The design makes the following assumptions:

(I) The design assumes a non-Byzantine scenario.

(II) The number of servers in a Paxos group is fixed.

(III) A server only fails by being unable to send or receive any message to or from any other parties.

(IV) A server recovered from a failure is in the state right before the failure.

(V) A client will submit a new request only after the previous request got replied by some server.

In essence, the assumptions (III) and (IV) forbid that following scenarios:

- Server/client a can receive message from server/client b but b cannot receive message from a.
- Server/client a can communicate with server/client c but server/client b cannot communicate with c.
- Server a has gone from state n to $n+1$ before experiencing a failure. After recovered from the failure, server a is back to state n or any earlier state.

The assumptions are reasonable by the nature of TCP and non-volatile computer memory.

3.2 Alive or Dead

Assumption (III) effectively restricts a server into two conditions:

- **Alive:** The server is available in the network. An alive server is reachable from all other alive servers/-clients, and all other alive servers/clients are reachable from the server.
- **Dead:** The server is unavailable in the network. A dead server is unreachable from all other servers/-clients, and all other servers/clients are unreachable from the server.

Note that the perception of a server being "alive" or "dead" may be different by individual servers. An alive server is perceived as alive by every other alive server but not the dead ones. A dead server with network failure perceives every other server as dead but itself as alive.

3.2.1 Ping

Every server in the Paxos group is broadcasting a `Ping Message` every `PingTimeout`[1]. The `Ping` serves two purposes:

- If a server a has not received any `Message` from another server b for two `PingTimeouts`, then b is determined dead by server a.
- The `Ping Message` contains the sender's knowledge about which `Slot` each server has executed in the Paxos group. This can help other servers garbage collect executed commands, which will be further discussed in §3.3.3.

3.3 Deterministic State Machine

The project implements a Paxos state machine. The state transition is caused by executing a command submitted by a client. Because the state transition is deterministic, the state of a server can be described as an ordered list of commands executed by the server. In our design, the list is made of `Slots` filled with commands. The decision of filling which command into which `Slot` is decided through Paxos algorithm. A `Slot` can be executed if and only if (1) all previous `Slots` are executed and (2) the majority of the servers have consensus on which command to be filled into the `Slot`. We say a server is at state n if it has executed the nth `Slot` and has not yet executed the $(n+1)$st `Slot`.

3.3.1 Up-to-date or Stale

Because of some failure, a server may be stuck at state n, while other servers have already got to state $n+d$. Once the server is recovered from the failure, by assumption (IV), it is now still at state n. Even though `PrepareReply` and `AcceptRequest` can sync the server with the latest state, if d is too large, it is infeasible for `PrepareReply` or `AcceptRequest` to help the server catch up. A constant `MAX_NUM_OF_COMMAND_PER_MESSAGE`[2] is defined such that if $d >$ `MAX_NUM_OF_COMMAND_PER_MESSAGE`, then the server is "stale"; otherwise, if $d \leq$ `MAX_NUM_OF_COMMAND_PER_MESSAGE`, then the server is "up-to-date". If a server is found to be stale, it is excluded from the Paxos group until some other server has helped it catch up to be up-to-date.

Note that the perception of a server being "up-to-date" or "stale" may be different by individual servers. Because each server may have a different latest state number as far as it has known, server a may be perceived as up-to-date by server b but not by server c.

3.3.2 Recovering

When a server is stale, other servers will try to recover it to the latest state. To do this, another server will send a `Recover` message to the stale server containing the commands it has missed. The number of commands in one `Recover` message is limited by `MAX_NUM_OF_COMMAND_PER_MESSAGE`, so if the stale server has missed too many commands, these commands will be divided into batches and send to the stale server one by one.

Sending the `Recover` message requires the sender to copy a large number of commands, which is an expensive operation. Thus, any alive server is responsible to recover exactly one other alive server. The responsibility is decided by forming all non-leader alive servers as a ring by their addresses. Each non-leader alive server is responsible for the next higher addressed non-leader alive server. The reason why leader is not involved in this ring is to ensure the liveness of the leader. Also, the leader election rule (3.1) introduced later guarantees that leader is always up-to-date, so it does not need to be recovered. However, one exception is when leader is

[1] currently set to 100ms.
[2] currently set to 10k.

the only one up-to-date among all alive servers. In such case, the leader will begin to recover the next higher addressed alive server.

3.3.3 Garbage Collection

An implication of assumption (IV) is that if every server has executed the nth `Slot`, then any info about nth `Slot` is no longer needed. In fact, the only reason why we need to remember what is the command chosen for a `Slot` is to inform the servers who does not know. Thus, in order to save memory space, if a server finds out every one has executed the nth `Slot`, it will then garbage collect any `Slot`s from the 1st to nth one still in memory.

3.4 Leader Election

The design uses a leader election mechanism to ensure liveness. Only the leader can process the requests from clients, send `PrepareRequest`s, send `AcceptRequest`s, and reply to clients with results. There is exactly one leader recognized by the majority of servers if the majority of the servers are alive. In particular, server a is recognized as the leader by server b if and only if

$$a \text{ has the highest address among all servers that are alive and up-to-date by } b\text{'s perception.} \tag{3.1}$$

If a server finds itself satisfying (3.1) by its perception, then it promotes itself as the leader. Because of periodical `Ping`, aliveness and staleness are frequently syncing among all alive servers. It is unlikely to have two leaders among all alive servers for any time period longer than `PingTimeout`.

3.5 Paxos Protocol

The design follows the framework given in Lamport's paper but with some improvements. The complete life cycle of a command follows the following requests/replies:

- `PaxosRequest`: The client submits the command in a `PaxosRequest`. The client broadcasts the `PaxosRequest` to every server in the Paxos group.
- `PrepareRequest`: After receiving the `PaxosRequest`, the leader will broadcast a `PrepareRequest` to every other server in the Paxos group.
- `PrepareReply`: After receiving the `PrepareRequest` from the leader, a server will send a `PrepareReply` back to leader saying either accepting the proposal or rejecting the proposal.
- `AcceptRequest`: If the leader has majority of the servers accepting the proposal in `PrepareReply`, it will broadcast an `AcceptRequest` with the client command.
- `AcceptReply`: After receiving the `AcceptRequest` from the leader, a server will similarly send an `AcceptReply` back to leader saying either accepting or rejecting.
- `PaxosReply`: If the leader has majority of the servers accepting in `AcceptReply`, it will execute the client's command and send the client `PaxosReply` with the result.

3.5.1 Batched Commands

The design runs a multi-instance Paxos algorithm in a batched fashion. Suppose the leader is at state n. A single instance of Paxos algorithm will decide the commands from $(n+1)$st to $(n+1+d)$th `Slot`. Typically, the commands are the `pendingCommands` as far as the leader has received right before sending the `AcceptRequest`. Until this instance of Paxos succeeds or be abandoned, the leader will not initiate another instance. In the meantime, all newly received commands from clients enter `pendingCommands`, which will be decided during the next instance.

All commands received by the leader are organized in three data structures:

- `executed`: This is an ordered list of commands that majority of the servers have reached consensus on and hence executed. While the number of commands in `executed` may decrease as a result of garbage collection described in §3.3.3, the data structure keeps track of the `Slot` numbers of the commands. If the `Slot` number of the last command is $n-1$, then the server is at state n.[3]
- `uncertain`: This is an ordered set of commands, but unlike `executed`, these are the proposed commands for `Slot`s immediately after the end of `executed` that no majority of the servers have reached consensus on. Suppose the last `Slot` of `executed` is numbered $n-1$, and `uncertain` has size d, then the commands in `uncertain` are the proposed commands for `Slot`s n to $n-1+d$.
- `pendingCommands`: This is an unordered set of commands. When previous batch has been proposed but not yet settled, all newly received commands will be collected in `pendingCommands` to be proposed during the next Paxos instance.

3.5.2 PrepareRequest & PrepareReply

PrepareRequest `PrepareRequest` contains a proposal number: `<num,address>`. The `address` is always equal to the leader/sender. This ensures that every leader sends a distinct proposal number. The comparison between any two proposal numbers trivially follows lexicographical order. When a leader sends `PrepareRequest`, it will adjust the `num` field to ensure this is the highest proposal number as far as it has known.

PrepareReply When received `PrepareRequest`, the server will firstly check whether the sender is indeed the leader and up-to-date by the server's perception. If any of these two requirements fails, the server will simply ignore this `PrepareRequest`. If both requirements are satisfied, the server will compare the proposal number against the highest one as far as it knows. There are two cases:

- *The received proposal number is higher than the highest one.* The server will accept this proposal. In Lamport's paper, the acceptor is required to respond with the value v of the previous highest-numbered proposal. In our case, the value v is the combination of `executed` + `uncertain`.
- *The received proposal number is lower than the highest one.* The server simply rejects the proposal by replying with its `maxProposalNum`, which is higher than the received proposal number in this case.

3.5.3 Skip PrepareRequest

After a `PrepareRequest` is accepted by majority of the acceptors, an optimization is to allow the leader issuing multiple rounds of `AcceptRequest`s without `PrepareRequest`s until some `AcceptRequest` is rejected by some acceptor. The `AcceptRequest` number is in the form `<num,address,round>`, where the first two fields are from the accepted proposal number, and `round` is the round number of the `AcceptRequest`.

One way to think about why we can skip the `PrepareRequest` is that when the leader at state n issues a `PrepareRequest`, the leader is issuing a proposal number for `Slot`s n to ∞. Thus, subsequent `AcceptRequest`s are simply giving values to `Slot`s that have already passed the phase 1 in Lamport's paper.

3.5.4 AcceptRequest & AcceptReply

AcceptRequest When received `PrepareReply`, the leader will check the `maxProposalNum` field. If it is higher than its proposal number, then the leader will initiate a new `PrepareRequest`. Otherwise, the leader considers the proposal accepted. After majority of the acceptors reply with proposal accepted, the leader then sync `executed` + `uncertain` to the `PrepareReply` with highest `maxAcceptedNum`. Note that since the first two fields of `maxAcceptedNum` are borrowed from the corresponding proposal number. The highest `maxAcceptedNum` is guaranteed to come from the highest-numbered proposal.

[3]The Slot number starts from 0, so the Slot numbered $n-1$ is the nth Slot.

The leader then sends `AcceptRequest`. The leader appends all `pendingCommands` to the end of `uncertain` it got from `PrepareReply`. The `AcceptRequest` contains the new `executed` + `uncertain` representing the chosen values of leader. The `AcceptRequest` number `<num,address,round>` is described in §3.5.3.

AcceptReply If the acceptor has not received a higher-numbered proposal, it syncs with the `AcceptRequest`'s `executed` + `uncertain` and reply with acceptance; if the acceptor has received a higher-numbered proposal, it simply reply with the higher `maxProposalNum` to indicate a rejection.

Reply Client After `AcceptRequest` is accepted by the majority of the acceptors, the leader then execute all commands in `uncertain`, since these values have the consensus of majority. The execution results are sent to the clients in `PaxosReply`.

3.6 Lock Service

The lock service in this implementation supports three operations: `lock`, `unlock`, and `query`. The design supports `lock` and `unlock` multiple locks at the same time. Suppose client submits a request to `lock` a set of locks U, then the rule of `lock` is as followed:

(a) If the client is currently holding some lock not in U, then the `lock` request fails;

(b) If any lock in U is locked by some other client right now, then the `lock` request fails;

(c) Otherwise, the `lock` request succeeds.

Suppose client submits a request to `unlock` a set of locks V, then the rule of `unlock` is as followed:

(a) If any lock in V is locked by some other client right now, then the `unlock` request fails;

(b) Otherwise, the `unlock` request succeeds.

The `query` request always succeeds, and it returns the locks currently held by the client.

3.6.1 Deadlock

This design effectively eliminates the possibility of a deadlock. The deadlock situation is when client a holds lock 1 and wants to acquire lock 2, while client b holds lock 2 and wants to acquire lock 1. In our design, a client is not allowed to acquire new lock when he or she has already held some lock. When client a or b wants to acquire both lock 1 and 2, he or she can only submit one `lock` request to acquire both locks in one transaction. Whoever has his or her request executed first got both lock 1 and 2.

4 Implementation

4.1 Overview

There are two parties in the implementation: `PaxosServer` and `PaxosClient`. Each party is implemented as a Java subclass of `Node`. Especially, `PaxosServer` and `PaxosClient` contain handler methods to be called when (1) a message is received from either client or other Paxos server, or (2) a timeout set previously is triggered. The implementation of `Node` manages all the network I/O and timeouts. It asynchronously call the handlers in `PaxosServer` and `PaxosClient` when the corresponding events are triggered.

4.2 Message

`Message` is a Java interface implemented by all requests, replies, and additional messages like `Ping` and `Recover`. The communication between any two `Nodes` is the exchange of `Messages`. The `Message` interface is also a subinterface of Serializable, `Logged`, and `Copyable`. Because `Message` needs to be serialized and deserialized in network I/O, `Serializable` is necessary. `Logged` is for log purpose. `Copyable` requires any class implementing `Message` to have a `immutableCopy()` method, which returns a copy of the `Message` with a promise not to modify anything in the copy in the future. The necessity of `Copyable` will be discussed in §4.7.1 under section Concurrency.

4.3 Timeout

`Timeout` is a Java interface. It contains a single method `timeoutLengthMillis()`, which tells `Node` what time in future to call the `Timeout`'s handler in `PaxosServer` or `PaxosClient`. The `Timeout` implementation serves two purposes: (1) to periodically perform some operations like `Ping`ing other servers, and (2) to resend some request in case the request is dropped due to network failure or some other reasons.

4.4 Node

The design logic of `Node` is to handle all the network I/O and make asynchronous calls. Thus, its subclasses `PaxosServer` and `PaxosClient` can focus on the Paxos protocol. When a `Message` arrives at `Node`, it will call its subclass' handler method through Java reflection. In particular, for a `Message` with class `MessageClass`, it will call a method with signature `handleMessageClass(MessageClass, Address)`. The `MessageClass` parameter is the message, and the `Address` parameter is the address of sender. For example, if `Node` receives a `PrepareRequest`, it will call the method `handlePrepareRequest(PrepareRequest, Address)`. Similarly, for `Timeout`, if a `Timeout` is triggered, the `Node` will call a method with signature `onTimeoutClass(TimeoutClass)`. It is the responsibility of `Node`'s subclasses to implement such methods for all `Messages` and `Timeouts` that are possible for `Node` to receive and trigger.

4.4.1 ArrayBlockingSetQueue

`ArrayBlockingSetQueue` is a data structure implemented for producer-consumer queues. It is a subclass of ArrayBlockingQueue in Java standard library. A BlockingQueue, defined by Java standard library, is a queue that gives producer/consumer an option to wait until it is possible to enqueue/dequeue an element. `ArrayBlockingQueue` is a `BlockingQueue` with limited capacity, and `ArrayBlockingSetQueue` additionally ensures that all elements in `ArrayBlockingQueue` are distinct. The `ArrayBlockingSetQueue` data structure serves two purposes:

- Used by `ConnectionPool` as `outboundPackages` queue to store any `Messages` submitted by `Node` but not yet send out by any one of the `SendTask` threads of `ConnectionPool`. In such scenario, `Node` is the producer, and `SendTask` threads are the consumers.
- Provided to the constructor of ThreadPoolExecutor as an internal structure to store `Message` handling tasks submitted but not yet executed by thread pool.

The main reason why we need `ArrayBlockingSetQueue` instead of provided `ArrayBlockingQueue` is to ensure the queues have distinct elements. After all, it is undesirable for `ConnectionPool` to send duplicated `Messages` or for `ThreadPoolExecutor` to handle duplicated `Messages`. In addition, for `ConnectionPool`, the `ArrayBlockingSetQueue` implements an additional feature to discard the oldest `Message` when enqueueing a new `Message` to a full queue. The underlying logic is that the info of old `Message` may already be out-of-date. It is more reasonable to keep the new `Message` in queue during network congestion.

4.4.2 ConnectionPool

`ConnectionPool` is implemented for `Node` to handle network I/O. When `Node` wants to send some message to another `Node` or `Node` has accepted an incoming connection through listening, it creates a `ConnectionPool` dedicated to the network communication between it and the other `Node`. If a `Node` is communicating with ten other

Nodes, then it maintains ten ConnectionPools.

ConnectionPool usually maintains a number of TCP connections. Each connection has a dedicated SendTask thread and a ReceiveTask thread:

SendTask The SendTask thread continuously "poll"s Packages from the outboundPackages queue. Each Package is a wrapper of one Message. The queue is a ArrayBlockingSetQueue, and its "poll" method lets the caller thread to dequeue an element if available or wait up to a specific timeout. In SendTask, this timeout is a constant TEST_ALIVE_INTERVAL[4]. If the "poll" timeouts, SendTask will send a special TestAlive Message. The reason why SendTask does this is to ensure that the SendTask sends a message through the connection every TEST_ALIVE_INTERVAL. Thus, if a connection fails, the ConnectionPool of both sides can quickly discover the failure by setting SO_TIMEOUT.

SendTask sends the Package using the object serialization provided by Java standard library. The Package is firstly written to an ObjectOutputStream, which outputs to a BufferedOutputStream and then to the socket OutputStream. The reason why we use a BufferedOutputStream in the middle is due to a finding that using a BufferedInput/OutputStream between ObjectInput/OutputStream and socket Input/OutputStream brings order of magnitude increase in network I/O efficiency[5]. SendTask flushes ObjectOutputStream every time it writes a Package to ensure timely delivery.

A special case of SendTask happens when the connection is the first connection created by the Node. In such case, the other side of the connection only knows the IP address of this Node but not the port number. This brings a problem since all Nodes are addressed by the combination of IPv4 address and listening port number. Thus, in such case, the very first two bytes written by SendTask will be the listening port number of this Node. In the meantime, when a Node accepts a new incoming connection, it will read the very first two bytes as the port number of the creator of connection.

ReceiveTask The ReceiveTask thread continuously read from the connection. Like SendTask, it uses a ObjectInputStream ← BufferedInputStream ← SocketInputStream design. Every time ReceiveTask reads a Message, it calls a handler from Node, which will assign a work thread to call the corresponding Message handler in Node's subclass. Thus, ReceiveTask does not need to wait until the Message is handled. It can immediately get back to reading new Message from the connection. As mentioned in SendTask, ReceiveTask sets the SO_TIMEOUT to a constant TEST_ALIVE_TIMEOUT[6]. If the read operation of ReceiveTask waits more than TEST_ALIVE_TIMEOUT, it determines that the connection fails and then orderly close the connection. Closing the connection causes corresponding SendTask to experience write failure and thus terminates SendTask thread.

ConnectionCreationTask Besides the SendTask thread and ReceiveTask thread for each TCP connection in ConnectionPool, the ConnectionPool also has a long-lived ConnectionCreationTask thread running from the construction of ConnectionPool to its closure. ConnectionCreationTask does only one task: to create new TCP connection when the number of connections is lower than constant MIN_NUM_OF_CONNECTIONS[7] or outboundPackages is full. If none of these two criteria is met, then the thread enters wait. The thread is notifyed every time a connection is closed or outboundPackages is full.

Besides some network advantages of concurrent TCP connections, there are two particular reasons why we need multiple connections in ConnectionPool:

- When two Nodes want to connect to each other at the same time, there will be two TCP connections created. If we do not allow multiple connections, then dropping which one would be a problem.

[4]current set to 500ms.
[5]https://stackoverflow.com/questions/23506651/why-is-inputstream-readobject-taking-so-much-of-time-reading-the-serialized-obje
[6]currently set to $3 \times$ TEST_ALIVE_INTERVAL $=$1500ms.
[7]currently set to 10.

- The sending and receiving involve object serialization and deserialization. These operations are expensive, so by having multiple `SendTask` and `ReceiveTask` threads, one can parallel the workload to increase network I/O efficiency.

However, creating too many connections also brings problems. Thus, we limit the number of connections to be less than or equal to a constant `MAX_NUM_OF_CONNECTIONS`[8].

4.5 PaxosServer

`PaxosServer` has the implementation of all Paxos protocol. In particular, it contains the callback handlers for the requests and replies described in §3.5. In this section, we will be talking about the implementation of parts in `PaxosServer`.

4.5.1 PaxosServer.Slots

`PaxosServer.Slots` is the underlying data structure of `executed` mentioned in §3.5.1. `PaxosServer.Slots` uses ArrayDeque to store commands in `executed`, and it keeps the `Slot` number of the very first command in `ArrayDeque`. Thus, it knows the `Slot` number of every command in `ArrayDeque` even if commands before the first one has been garbage collected. Every time `PaxosServer` executes a command, it is enqueued into the `ArrayDeque` of `PaxosServer.Slots`. If the `PaxosServer` discovers every server has executed some `Slot`, then any command before and in the `Slot` will be removed from `ArrayDeque`. Note that by design, if a `PaxosServer` has executed the command in `Slot` n, then it has executed all commands from `Slot` 0 to n.

4.5.2 Application

`Application` is where the service logic implemented. In our implementation, `Application` is a Java interface, which provides a method `Result execute(Command)`. Each request received from the client contains a `Command` to be executed by the `Application`, and the returned `Result` will be in the reply to client. In this project, the lock service described in §3.6 is implemented as a `LockApplication`.

AMOApplication `AMO` stands for "at most once". `AMOApplication` is a wrapper over `Application`. By assumption (V), the client can assign a strictly increasing `sequenceNum` for each command he or she submits. Once `AMOApplication` has let the `Application` execute a command from a client with `sequenceNum` x, then it ensures that any command submitted by the same client with `sequenceNum` less than or equal to x cannot be executed by `Application`. In particular, `AMOApplication` stores results of the latest commands submitted by all clients. Thus, when `PaxosServer` receives a command `Application` has already executed, `AMOApplication` can prevent `PaxosServer` from going through the Paxos algorithm for this command again and directly reply the client with result if the command is the latest one. Note that if the command is not the latest one, by assumption (V), the client has already got the result of the command.

4.5.3 PaxosServer.Leader

`PaxosServer.Leader` is a data structure to maintain the leader state when the `PaxosServer` is the leader of the Paxos group. For example, `PaxosServer.Leader` records the `pendingCommands` mentioned in §3.5.1 and addresses of the servers who has not yet replied `PrepareRequest` or `AcceptRequest`.

4.5.4 PaxosServer.Acceptor

`PaxosServer.Acceptor` is a data structure to maintain the acceptor state. It is maintained at all time since every server is always an acceptor even if it has been elected as the leader. It has only two fields `maxProposalNum` and `maxAcceptNum`.

[8] currently set to $4 \times$ MIN_NUM_OF_CONNECTIONS $= 40$

4.6 PaxosClient

`PaxosClient` has the implementation of a client to submit `PaxosRequest` to `PaxosServers`. It follows assumption (V), and assign strictly increasing `sequenceNum` to commands it submits. If a request is not replied for time period longer than `ClientTimeout`[9], then a `ClientTimeout` will be triggered, and `PaxosClient` will resend the `PaxosRequest`.

4.7 Concurrency

The implementation uses multiple tools of Java to handle concurrency issues. For `PaxosServer` and `PaoxsClient`, because only the handler methods for `Messages` and `Timeouts` will be called, these methods use `synchronized` keyword to ensure proper locking and unlocking.

For data structures such as `Map` and counters in `Node` and `ConnectionPool`, the implementation conveniently uses ConcurrentHashMap, AtomicLong, and LongAdder from Java standard library to ensure thread safety.

For `ArrayBlockingSetQueue`, since it is a subclass of `ArrayBlockingQueue`, the implementation uses Java reflection to access the parent's ReentrantLock. In every method, `ArrayBlockingSetQueue` locks the parent's `ReentrantLock` and release it at the end of the method. Because `ReentrantLock` can be locked by the same thread multiple times, superclass' methods are unaffected when `ArrayBlockingSetQueue` calls them while holding the lock.

4.7.1 Copyable

As mentioned in §4.2, `Copyable` is an interface to force `Message` to provide an immutable copy of itself. To see why `Copyable` is necessary, consider a scenario when `PaxosServer` wants to send a `Message`. `PaxosServer` calls the `send` method in `Node`, who simply enqueues the `Message` to an underlying queue and then returns. The `SendTask` thread of some `ConnectionPool` will later dequeues the `Message` and perform network I/O. This implementation requires the `Message` to be unmodified while it is in the queue; otherwise, the `Message` eventually send is different from what `PaxosServer` intended. However, the `Message` usually contains some internal state of `PaxosServer` like `executed` that may be changed by `PaxosServer` thread right away. It is also undesirable to make `PaxosServer` thread blocked until the `SendTask` finishes network I/O. Thus, the solution is to let the `send` method in `Node` call the `immutableCopy()` to enqueue an immutable copy of the `Message`.

4.7.2 Thread Pools

The implementation uses multiple ThreadPoolExecutors from Java standard library to realize a thread pool design. In addition, the implementation uses ScheduledThreadPoolExecutor to realize the triggering of `Timeouts` in future. Thread pools are carefully tuned to ensure high throughput of the system.

PaxosServer For `Message` handling tasks, `PaxosServer` uses a `ThreadPoolExecutor` with thread pool size fixed to a constant `MESSAGE_HANDLER_EXECUTOR_NUM_OF_THREAD`[10] and `ArrayBlockingSetQueue` as the `workQueue` of `ThreadPoolExecutor` with fixed capacity `MESSAGE_HANDLER_EXECUTOR_QUEUE_SIZE`[11]. The reason why `PaxosServer` uses a small fixed size thread pool is that all `Message` handler methods are marked by `synchronized` Java keyword, so the locking forbids `PaxosServer` from having many threads working together. As for fixed capacity `workQueue`, when there are too many `Messages` waiting to be handled, the design allows `PaxosServer` to simply ignore some `Messages` until workload spike passes. Because unlimited `workQueue` never ignores or rejects a `Message` handling task, it may affect the responsiveness of `PaxosServer`, when more and more `Messages` are built up in the queue. Under some test situation, the queue can get even long enough to take all of JVM heap memory.

[9] currently set at 100ms
[10] currently set to 5.
[11] currently set to 100.

Node The implementation of `Node` uses two thread pools for its tasks: a `ThreadPoolExecutor` with an unlimited thread pool size and a `ScheduledThreadPoolExecutor` with thread pool size fixed to 5. `Node` uses `ThreadPoolExecutor` to finish tasks like logging, `ConnectionPool` garbage collection, and adding incoming connection to `ConnectionPool` when accepted from listening port. The `ThreadPoolExecutor` is also passed to `ConnectionPool` to create threads like `SendTask` and `ReceiveTask`. The threads of `Node` are largely independent and equally important, so there is no need to tune `ThreadPoolExecutor` in `Node`. `ScheduledThreadPoolExecutor` is used to handle all `Timeouts` and periodical logging. It has a small size thread pool because what it does is simply assigning the work to `ThreadPoolExecutor` when it is time.

5 Deployment

The project is implemented in Java 8 and tested in ARM version Ubuntu 20.04.3 LTS virtual machines. Besides Java standard library, the project also uses Apache Commons Lang 3.12.0 and Lombok 1.18.12.

To properly run the project, `PaxosServer` and `PaxosClient` require a config file with IPv4 addresses of all `PaxosServers`. For example, to setup a Paxos group of 3 servers on local machine listening to port 2000, 3000, and 4000 respectively, one needs to create a `server_ips.config` file:

```
127.0.0.1:2000
127.0.0.1:3000
127.0.0.1:4000
```

The commands to run `PaxosServers` are:

```
java -jar paxos_server.jar 127.0.0.1:2000 server_ips.config
java -jar paxos_server.jar 127.0.0.1:3000 server_ips.config
java -jar paxos_server.jar 127.0.0.1:4000 server_ips.config
```

The command to run a `PaxosClient` using port 5000 on local machine is:

```
java -jar paxos_client.jar 127.0.0.1:5000 server_ips.config
```

6 Test

The Paxos protocol of this project directly uses JUnit tests of DsLabs for testing. Besides, the project also includes a `TestClient` to test in distributed runtime environment. The `TestClient` works by continuously generating random `LockCommands`. It executes the commands on local `LockApplication` while sending the commands to Paxos group. It is expected that the results returned by the Paxos group should be identical to the results returned by `TestClient` executing commands locally. Similar to `PaxosServer` and `PaxosClient`, the command to run `TestClient` using port 5000 is:

```
java -jar test_client.jar 127.0.0.1:5000 server_ips.config
```

7 Future Improvements

Due to the limited time of this project, there are some improvements to the project considered but not implemented by the author.

7.1 Copy and Serialization

The workflow of `Node` sending a `Message` is to (1) obtain an immutable copy of the `Message`, (2) enqueue the immutable copy to queue, (3) dequeue the copy, and then (4) serialize it to send through socket's `OutputStream`. However, serializing the `Message` has already copied the object in some sense. An improvement is to directly serialize the `Message` into bytes and enqueue the bytes into the queue. Thus, the `SendTask` thread can simply

dequeue the bytes and send through socket's `OutputStream`. However, the problem unsolved is that the object serialization can only been done by creating an `ObjectOutputStream`. Every time an `ObjectOutputStream` is created, it writes an initial header to underlying `OutputStream`. When the `ObjectInputStream` tries to deserialize the stream of bytes, it only expects the initial header at the very beginning. If every object contains an initial header, the `ObjectInputStream` will throw `Exceptions`.

7.2 Concurrency of PaxosServer

The current implementation of `PaxosServer` uses Java `synchronized` keyword to ensure thread safety. This is the most naive design, which only allows threads to finish their works one by one. A more desirable and complicated design is to add locks for individual data structures. One may also use read/write locks to enhance parallelism.

7.3 Write Commands to Disk

In §3.3.3, we mentioned that if all servers have executed the command in `Slot` n, then the command is garbage collected and removed from memory. Thus, the `PaxosServer` can continuously process commands without running out of memory. In some cases, when some server is down for too long, the number of commands in alive servers is continuously built up. To prevent running out of memory, the current design stops the servers from processing new commands if the number of commands in memory exceeds some point. At that time, only if the failed server is recovered, the commands in servers can be garbage collected and then processing new commands. In such situation, one may want the alive servers to continue processing commands while storing the commands in disk. When the failed server is alive again, the other servers can read commands from disk to help the failed server recover. Furthermore, one may want all the commands written to disk for the purpose of logging or debugging.