Nicholas Shahan
Eric Zeng

# CSE 550 - Paxos Lock Service Writeup

- CSE 550 - Paxos Lock Service Writeup
  - Overview
  - Related Works
  - Paxos
    - Usage (for applications)
    - Paxos Struct
    - RPC Endpoints
    - Proposer's Protocol Description
    - Acceptor's Protocol Description
  - LockService
    - LockService Struct
    - RPC Endpoints
      - Lock
      - Unlock
    - Internal Functions
      - enqueueRequest
      - dequeueRequests
      - getAgreement
      - commitOperation

## Overview

We implemented a Paxos-based Lock Service in Go. We started with a generic Paxos implementation, and built a lock service application that used Paxos to guarantee a consistent ordering of lock and unlock operations. We also wrote a client to connect to the lock service. Both the client and service application can be launched via command line or other Go applications. We created small mainline programs to manually demonstrate the service from the command line.

## Related Works

Our implementation of Paxos was derived from a similar Paxos project in CSE 452/552M, the undergrad/5th year masters distributed systems class. Both of us took the 552M class in Winter 2015, and completed the Paxos implementation independently. For this assignment, we modified the networking code to use network sockets instead of Unix sockets, so that the system can be distributed across different machines.

Nicholas Shahan
Eric Zeng

The Paxos protocol logic is based on pseudocode found here:
http://nil.csail.mit.edu/6.824/2015/notes/paxos-code.html

# Paxos

We built a generic Paxos library in Go that is intended to be used by a server-side application. The code is located in `src/lockservice/paxos.go`. The intended use case for our library is for distributed applications to use Paxos to agree upon a consistent order of operations, one at a time. We represent this internally using an in-memory log of instances.

## Usage (for applications)

These are the methods we expose to the library user:

| Method | Description |
|---|---|
| Start(seq int, v interface{}) | Call when you want to persist the value v in the Paxos log, at instance number seq. |
| Done(seq int) | Marks all instances less than or equal to seq as done, meaning Paxos can garbage collect them safely. |
| Max() int | Returns the highest instance known to this Paxos peer. |
| Min() int | Returns the lowest instance known to all peers in the cluster. |
| Status(seq int) (bool, interface{}) | Checks the status of instance seq. Returns false if the instance hasn't been decided yet, returns true and the decided value if it has. |
| MakePaxos(peers []string, me int) | Constructs a new Paxos object. |

To add a value to the Paxos log, typically the application will do the following:

First, the application calls Start() to kick off a consensus round. The instance number (seq) should be one greater than the highest decided instance the application knows about. That means it will have to keep track of the max instance it has seen. The value (v) is the value it wants to persist.

Next, the application should call Status() on the instance it just Started to check if it has been decided. Since the Paxos protocol will take some time to run, Status() may initially return false. Therefore, it should loop (with an increasing backoff to not starve the Paxos thread) and call Status() until it returns true.

Once Status returns true, it means that the Paxos group has decided the value of the instance, and returns that as well. The application can then make changes to its own state based on that value.

However, that value might not be the same value as the one the application called Start() with. If the values are not the same, the application needs to start over and call Start() with the next instance. But even if they are not the same, the application still needs to apply the instance to its state, or else its state won't be consistent with the other application nodes!

## Paxos Struct

This struct stores all of the internal state of a Paxos node. None of the fields should be accessed by the application.

| Field Name | Type | Description |
|---|---|---|
| mu | sync.Mutex | This lock is used to enforce mutual exclusion during RPC calls. The RPCs will be described later. |
| peers | []string | Strings representing other nodes in the paxos cluster. Peers should be formatted as <ip>:<port> or :<port> for local peers. |
| me | int | The index of this Paxos instance in Paxos.peers |
| instances | map[int]*InstanceInfo | The log of instances agreed upon by the system. The key is an integer representing the instance number, and the value is an InstanceInfo struct, which contains the value decided in the instance. |
| min | map[string]int | A map of which instances have been committed by each node, and no longer need to be stored. Used if we want to garbage collect the instance log. Maps peer to last completed instance number. |
| majority | int | The number of nodes required for a quorum. |

## RPC Endpoints

These are the endpoints used by Paxos nodes to communicate with each other. The application nodes shouldn't interact with these.

Nicholas Shahan
Eric Zeng

| Endpoint | Args | Reply |
|---|---|---|
| Prepare | Instance int, Proposal int | Err string, HighestAccept int, HighestAcceptVal interface{}, DecidedVal interface{}, Done int |
| Accept | Instance int, Proposal int, Value interface{} | Err string, AcceptedProposal int, DecidedVal interface{}, Done int |
| Decided | Instance int, Value interface{} | Done int |

## Proposer's Protocol Description

The proposer's protocol is executed by a Paxos node when the application wants a new instance to be added to the log. The code path starts when the application calls Start(), which does some initial error checking and initializing missing values. Then, it calls the private method propose() in a new goroutine to execute in parallel.

The node first sends out Prepare messages to the other nodes in the system, which consists of the instance number and proposal number. Then, it waits for a majority of nodes to respond with PrepareOk, indicating that they chose its proposal number. If it does not have get a majority to agree on its proposal, it tries again with a higher proposal number.

On the PrepareOk messages, the remote nodes return the highest Accepted values - so if other proposals for that instance have been accepted already, the node can learn the value they have all agreed upon.

Then, the node sends out Accept messages to other nodes in the system, which includes the instance number, proposal number, and value to be stored. The value to be stored is the value the application wants stored, unless another node has already had their value accepted. In that case, the node would have received that value in the PrepareOk message, and replaced its own value with the other node's value. This ensures that for every instance, they only Accept a single value.

Once the node has received a majority of AcceptOk messages, it means that a majority of nodes accepts that as the value of the new instance. Then, it broadcasts a Decided message to every node, which causes each node to write the instance in the Decided message to their instances log.

If at any point the node contacts a node which has Decided on the instance, it receives a reply indicating that the instance has been decided. Upon receiving this reply, it modifies its own

instance log, and broadcasts Decided messages to other nodes, ensuring that everyone has persisted the instance, even if some network links have been broken.

### Acceptor's Protocol Description

The acceptor's protocol describes how the Paxos node handles RPC messages from other Paxos nodes that are trying to propose and get agreement on instances. These messages are Prepare, Accept, and Decided.

The Acceptor is responsible for managing the log of instances. We represent each instance as a struct that contains various bits of state. The log is a map from the instance number to the instance struct.

Each instance contains the highest proposal number seen for that instance, the highest proposal number accepted, the value corresponding to the highest proposal number accepted, and whether the instance has been decided.

The Prepare handler is called by other peers to send in a proposal number. It accepts the Prepare if it is the highest proposal number seen for that instance. To prevent nodes from proposing higher and higher numbers without agreeing on a value, it sends back the highest proposal sent to Accept, and its value (if one was accepted). The node should use that value as the value to Accept to ensure progress.

The Accept handler lets other peers set the value of an instance once it has Prepared and has learned a consistent value. If the proposal number is greater than or equal to the highest one it has seen, it sets the instance's highest proposal and value to the message's value. Every time a node calls Prepare for that instance, it will return that value, so that all nodes in the future can agree on the value.

The Decided handler is invoked when the node needs to be informed that the value of an instance has been decided. This flips the boolean Decided flag on the instance to true. After an instance is decided, any RPC calls to Accept or Prepare will simply respond with the value and an error indicating it is decided, which lets the proposer commit the value and move onto the next instance.

## LockService

We created a distributed lock service application that uses Paxos to ensure consistent state at all nodes. The lock service supports two operations, Lock(x) and Unlock(x), where x is a lock id.

Each node in LockService maintains a mapping of lock id to its lock status. The lock status can either be locked by a specific client (identified by its client id), or it can be unlocked (-1).

Nicholas Shahan
Eric Zeng

We determine how to modify the locks in a consistent manner across all by agreeing on all lock operations through Paxos. Client requests are received by the RPC handlers and added to a local request queue on each node, and one request is proposed to Paxos at a time.

The Paxos operation log is a serialized list of all lock and unlock attempts made by the LockService nodes. This means the log contains an ordering of operations but does not mean all operations are valid. This is because in order to determine a consistent order for acquiring locks, all attempts to lock must be committed to Paxos.

The core component of the service is the `LockService` struct, as well as its methods. The RPC handler methods serve as the external API which clients can interact with.

## LockService Struct

| Field Name | Type | Description |
|---|---|---|
| locks | map[int]int | The state of all locks managed in the service. Maps lock id -> lock status. If unlocked, will be -1. If locked, will be the client id of the lock holder. |
| px | *Paxos | The paxos instance for this node. |
| max | int | The highest paxos instance number committed to the local state at this node. |
| requests | chan Request | Queue for client requests to be executed. |
| servers | []string | Strings representing other nodes in the LockService cluster. Peers should be formatted as <ip>:<port> or :<port> for local peers. |
| me | int | The index of this LockService instance in lockServcie.servers |

## RPC Endpoints

| Endpoint | Args | Reply |
|---|---|---|
| Lock | Client int, Lock int | Err Err |
| Unlock | Client int, Lock int | Err Err |

Nicholas Shahan
Eric Zeng

### Lock

When a client wants to lock a resource it calls the Lock RPC and identifies itself and the target lock. This will add the lock request to the local queue of pending requests to be submitted to Paxos.

Our lock service does not respond to the client until the lock operation has been committed to the local state which can only happen if the lock is unlocked. In the event that the lock is not available the result of the operation will be Requeue. The thread will sleep with an exponential backoff and then submit the request again. This will cause multiple instances of the same lock request to be sent to Paxos until the lock becomes available. We determined this to be the most correct way to determine a consistent ordering across multiple LockService nodes.

### Unlock

When a client wants to unlock it calls the Unlock RPC and identifies itself and the target lock. This request will be added to the same local queue and eventually submitted to Paxos. The unlock command will never need to requeue and once it is committed the RPC will return to the client.

In the case that Unlock is called on a lock that is already unlocked, or the lock is locked and does not belong to the client sending the Unlock request, the server will return an error message without blocking.

## Internal Functions

### enqueueRequest

Called by the the RPC handler threads to add a client request to the local queue. This method blocks while waiting for the response that is generated after sending the request to Paxos and attempting to commit the operation locally.

### dequeueRequests

Run by a single thread in the LockService node. Dequeues requests from the local queue and send them to Paxos by calling getAgreement. Returns the result of the operation to the original RPC handler.

### getAgreement

Attempts to form a Paxos consensus on the requested operation and commits the operation locally once it has been chosen. getAgreement will move the local state forward when this node is behind the others in the system. This is performed by learning that a different operation was already chosen for a given Paxos instance. In the event a different operation (proposed by another node) was chosen, the learned operation will be committed locally.

Nicholas Shahan
Eric Zeng

### commitOperation

Executes the actual changes to this LockService's local state. The changes to the state that are not valid will not actually be committed, and instead produce errors that are returned. This includes attempting to:

- Lock a resource that is currently held by another client.
- Unlock a resource that is already unlocked.
- Unlock a resource that is currently held by another client.

These errors are propagated back to the original RPC handler.

## Assumptions

We make a few simplifying assumptions in our implementation of Paxos and LockService.

We keep the log of instances in memory, unless all applications mark instances as Done using the API call. When those instances are marked Done, they are garbage collected and can never be recalled. Therefore, if the Done is used to clean up memory, we assume that crashed nodes never come back up, because they would lose all of the operations in the log from before they crashed.

If the application does not mark instances as Done, then we never garbage collect instances, and nodes can come back up. The node will run correctly until the instance log runs out of memory, so we assume that the Paxos instances don't run long enough for that to ever happen.

By making these assumptions, we avoid needing to implement a state transfer protocol for LockService so that it can garbage collect instances and support rejoining nodes.

For RPC calls, we assume that nodes or clients send at most one message. If the message drops, it just drops and the node or client won't automatically retry until it gets a positive response.