Haxos: A Distributed Lock Service in Haskell

Eugene Perederey Ravi Sankar Chris Pak

December 15, 2011

1 Introduction

The idea for this project stems from Frangipani, a distributed file system designed to be easy to use and highly scalable [2]. While we did not attempt to replicate this system in its entirety due to time constraints, we did implement several of the important pieces of this system. The primary focus was on creating a working implementation of Paxos.

Our system is a distributed lock service, entirely written in Haskell. This system is capable of dealing with contention for generic shared resources, though it is targeted at files on a virtual disk. A FUSE-based client can present that virtual disk as a local disk to users and enforce communication with the lock server prior to direct access of remote files. The distributed lock service itself consists of a primary lock server and several backup servers which can reconfigure themselves in the event of failure, using the Paxos algorithm, to continue to fulfill client requests without hidden data loss.

Outline The remainder of this article is organized as follows. Section 2 gives a brief review of how the Paxos algorithm works, and its relevance to the lock server we created. In Section 3, we explain the interface of the lock server from both the perspective of the client and the server itself. Finally, the details of the protocol are in Section 4.

2 Paxos

2.1 Algorithm Summary

Paxos is a protocol designed to enable multiple machines to reach consensus on a proposed value, even with unreliable machines or networks [1]. The value being "debated" often represents which machine will act as a primary server when any one could be chosen or which machine's existing state will represent the canonical version. We will provide a brief overview of Paxos here, but for details see [1].

During normal operation, there is one primary server and at least two backup servers. A client sends any requests for the service to the primary. After logging the request, the primary forwards the requests to the backups, which should all log the request as well before sending an ack back to the primary. When the number of machines that have logged the request (primary + backup acks) represents the majority of the machines, the primary can

execute the request and reply to the client. Thus, in normal operation, the primary server does all the work it would normally do to respond to a client request, plus some additional work to make sure that most of its backups have a copy of the requests that it has executed [1].

If things never went wrong, though, we wouldnt need the Paxos protocol at all. If the primary never receives an ack from one or more of its backups or one of the backups stops receiving messages from the primary and suspects the primary has crashed, a view change is required. This is true even though these conditions sometimes lead to a successful response to the client. The goal is to agree on a new view, which represents the members and state of the Paxos group.

Multiple proposers can simultaneously propose a new view. Every working Paxos server will receive these proposals and accept them if the proposed view represents newer information than its own current view and the other proposals the server has seen so far. When a proposer has received acceptance from the majority of the responding machines, it sends out an actual proposal that represents a new cohort, containing a majority of the previous group and/or the previous primary. This guarantees that the new view has all executed requests from the previous view. The members of this new view agree to join only if they havent already joined a view with a more recently committed request. The new primary is either the old primary or the machine with the most recently committed request. The group has now reconfigured itself and ensured its state is up-to-date, so it returns to normal operation, until another view change is called for.

2.2 Relevance for Lock Server

Paxos is used to establish consensus on a new view in response to failures, which can serve two purposes: if the primary is still online, it serves to correct the list of machines that actually know the list of locked resources and and the backups that we should expect responses from. If the primary is offline, it enables us to choose a new primary that will have a complete list of resources that have been locked so far. The view does not contain the request log or lock list itself, but by correctly choosing a new view and a new primary, we can guarantee that the new primary's lock list will be up to date.

3 Interface

3.1 Client

From the client's perspective, communicating with the lock service involves two steps. First, the client contacts a name server that provides the address of the current primary lock server, which is the only lock server the client should need to contact directly. We implemented a simple name server of our own, but another possible solution would be to add an entry with a short TTL to the local DNS server and then charge the current primary server with periodically updating the DNS server's record.

Second, the client sends a lock acquire or release request to that primary lock server (for more detailed information on the protocol, see Section 4). After sending a request, the client

will wait for a timeout period to receive either an approval or a rejection response. When it acquires a lock, it is free to access the shared resource (in this case, the remote file); as with mutexes, it is the client's job to enforce the locking system and only access resources that it holds the lock for, since the lock server does not act as an actual gate on the file server. If the client's request times out without a response, this likely means that the primary or several backups have failed, and the client can choose to retry its request after some time, when the Paxos algorithm has had time to either reassign the primary, or account for the lost backups. When a new primary is chosen, any uncommitted responses will be resent to the client, so a good client timeout period should be larger than the average time for a round of Paxos in a particular setup.

3.2 Server

Every lock server can operate either as a primary server or a backup server. A server's initial role and the port number on which it should listen are determined at startup by a config file, but the role can change at any time while the server is running. Each server, therefore, has the same running threads, some of which are dormant in certain roles and some of which have role-dependent behavior.

First, there is the client listener thread. The primary server listens for client requests to handle here; backups simply respond with the address of the primary server if they are contacted here. Second, there is the primary listener thread, which is only active for backups. The backups listen here for replicate requests from the primary and timed confirmations that the primary is still online. Third, the main thread listens for Paxos proposals for a view change. Finally, the Paxos reply listener is used by a Paxos proposer to listen for accept/reject messages sent by the main thread in response to view change proposals.

Our design currently uses UDP multicast for convenience when a lock server is making a Paxos proposal and when the primary is sending a replicate request to all the backups. This is easier than maintaining n^2 separate TCP connections, and it is probably usable on a reliable network with a sufficient number of backup servers. However, a production system would need to be modified to either use TCP connections or Paxos modified to provide Byzantine fault tolerance, as UDP makes no guarantees as to whether packets arrive in order or uncorrupted.

4 Protocol

4.1 Client / Server Interface

One important feature of a distributed system is hiding as much of the background work as possible and providing a simple interface to the client. In our system, there are only two data types associated with client side interaction. The first is a Request, the other is the server Reply. Both of these encode all of the necessary information required in a simple way as possible.

Client A client can request to acquire a lock on a path with read or write permission. If granted, the result will include a LockId. When the client releases a lock, they request to release the LockId, not the original path.

```
-- client commands
data Request = Acquire Permission Path
             | Release LockId
             | EmptyReq
               deriving (Show)

data Permission = ReadL | WriteL
type Path = ByteString
type LockId = Int
```

Server A server will either grant a lock request, returning a LockId, deny a lock request for a path, or return Retry, which indicates that the server could not guarantee a replicated/consistent response. The client can, therefore, retry later, allowing time for a round of Paxos to run.

```
-- server reply
data Reply = Grant LockId
           | Deny Path
           | Retry
           | EmptyReply
             deriving (Show)
```

4.2 Primary / Backup Interface

Whenever a primary lock server receives a request, it logs the request, then sends a ReplicateRequest to all of its backups. The backups log the request in turn, then send a BackupAck response to the primary. Once the primary receives a majority of the BackupAcks it expects, it can execute the request (i.e., create or release the lock) and respond to the client. If the primary does not receive sufficient BackupAcks, it initiates a round of Paxos. State is a data type used to hold all the state of each server.

```
--Primary -> Backup
data ReplicateRequest = ReplicateRequest { currentVS :: ViewStamp
                                         , clientRequest :: Request
                                         , commitedVS:: ViewStamp}
--Backup ->Primary
data BackupAck = BackupAck ViewStamp

data State = State {
      myId          :: CohortId
    , ts            :: ViewStamp -- last assigned timestamp (if primary)
    , role          :: Role      -- primary or backup
    , vcState       :: VCState   -- contains current view/last commit/etc.
    , phoneBook     :: PhoneBook -- how to find other cohorts
    , logPath       :: String
    , lockTable     :: LockTable
    , logs          :: Log
    , lastChangeTS  :: TimeStamp
    , paxosInitTID :: Maybe ThreadId -- backup thread to initiate Paxos
    }

data ViewStamp = ViewStamp ViewId TimeStamp
type CohortId = Int
```

4.3 View Change Interface

ServiceMsgs are sent by the View Manager in a round of Paxos. VCReplies are sent by View Underlings. When a round of Paxos is initiated, the View Manager sends a ViewChange. Recipients then respond with an AcceptViewChange or RejectViewChange. This all represents the Prepare phase of Paxos to select a new view_id.

Once the Prepare phase is complete, the View Manager sends a NewView message. Recipients respond with a NewViewRes, based on whether they have accepted a more attractive proposal already. This concludes the Propose phase to select the actual new view.

Finally, the View Manager notifies the new primary server of its role, and shares the message with backups to convert them back to active mode.

```
--These messages start and end View Changes
data ServiceMsg = ViewChange View ViewId --the View here is old_view
                | NewView ViewStamp View
                | YouArePrim View

--These messages are used in the process of negotiating a View Change
data VCReply = RejectViewChange View ViewId
             | AcceptViewChange ViewId CohortId Bool ViewStamp (Maybe View)
             | NewViewRes ViewId Bool
```

The actual View and view state are represented below.

```
--Normal operation, proposing, or listening?
data VCMode = VC_ACTIVE | VC_MANAGER | VC_UNDERLING

data VCState = VCState {
      vcMode :: VCMode
    , vcView :: View  -- last FORMED view
    , vcLatestCommitedTimeStamp  :: ViewStamp -- last committed operation
    , vcProposedViewID :: Maybe ViewId
    , vcRepliedCount :: Int                   -- proposing: phase 1
    , vcRepliedCount2 :: Int                  -- proposing: phase 2
    , vcRoundOneTS :: Maybe TimeStamp
    , vcRoundTwoTS :: Maybe TimeStamp
    , vcAcceptedMaybeView :: Maybe (View, ViewStamp) -- last accepted view
}

--Types defining Views
data ViewInfo = ViewInfo ViewId S.ByteString

data View = View { view_id    :: ViewId
                 , numCohorts :: Int

data ViewId = ViewId { proposer_cid :: CohortId
                     , view_id      :: ViewCounter}

type ViewCounter = Int
```

References

[1] D. Mazieres. Paxos made practical.

[2] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. *SIGOPS Oper. Syst. Rev.*, 31:224–237, October 1997.