CS240H: Implementing Paxos in Haskell

Jiayuan Ma
jyma@cs.stanford.edu

June 2014

Abstract

The Paxos algorithm is a distributed consensus algorithm, each round of which can be used to agree on a value proposed by a member of the group. The algorithm is tolerant to non-Byzantine errors (e.g. unreliable network environment and disconnected nodes). Despite the simplicity of the algorithm, implementing Paxos for real world applications in traditional languages can be involving and non-trivial[1]. This project implements the basic Paxos algorithm in Haskell, whose power of expressiveness can be helpful to simplify the implementation.

1 Introduction

This project implements the single-round basic Paxos algorithm in Haskell and simulates the behavior of the algorithm under a noisy network environment (with emulated packet loss and network delay). Section 2 gives a brief introduction to the Paxos algorithm we implemented. Section 3 discusses some Haskell design and implementation details. Section 4 focuses on some experience of implementing and simulating the protocol. Section 5 concludes the report.

2 Background

There are many Paxos or Paxos-like consensus algorithms described in the literature[1, 3, 4]. For this project, we focus on a widely-cited version of Paxos description by Leslie Lamport[2].
Suppose a group of machines or processes can communicate with each other through unreliable message passing and some members can leave the group nondeterministicly. We further assume non-Byzantine errors, where messages can be delayed and lost arbitrarily, but they are not corrupted once received. Since each machine or process can operate at arbitrary speed, stop or restart at any time, the algorithm assumes some information can be preserved persistently across crash and restart. The problem we want to solve here is to let this group agree on a single value one of its members propose. There are three different roles in the Paxos protocol (see the following). Each machine or process can play more than one role in the group.

Proposer Propose a value by sending messages to other member of the group.

Acceptor Accept proposed values.

Learner Learn whether the group has reached concensus in a particular round of the algorithm.

A single round of basic Paxos algorithm consists of two-phases (prepare phase and accept phase) as follows. Each phases includes one round of message sending/receiving.

Prepare Phase (i) Each of the proposers in the group select a proposal number n and sends a *prepare* request with number n to the acceptors in the system. Notice that the messages may not be received by all the acceptors. As long as the messages are received by a majority of machines (quorom), the algorithm can proceed.

Prepare Phase (ii) When an acceptor receives a *prepare* request with a proposal number n higher than that of any previous *prepare* requests it has responded to, it will respond to this *prepare* message with a *promise* message saying that "I'll not accept any more proposals numbered less than n and I've already accepted this highest-numbered proposal (if any) with the proposal number and the value $\langle n, v\rangle$". If an acceptor receives a *prepare* request with a proposal number n lower than that of request it has promised, it ignores those messages by not responding to them.

Accept Phase (i) If the proposer receives *promise* responses to its *prepare* requests (numbered n') from a majority of acceptors (quorom), then it sends an *accept* request to each of those acceptors a proposal number and value pair $\langle n', v'\rangle$, where v' is the value of the highest-numbered proposal among the responses, or can be any value if the responses reported no proposals. If the proposer does not receive *promise* responses from a majority of acceptors (quorom), then it would have to choose a higher proposal number n'' and restart. Notably, restarting prematurely can be inefficient, preventing other proposers' proposals from going through (see section 3).

Accept Phase (ii) If an acceptor receives an accept request for a proposal numbered n, it accepts the proposal and records the value unless it has already responded to a prepare request having a number greater than n.

3 Implementation

In this section, we will describe our design and implementation of basic Paxos algorithm in Haskell. We use the concatenation of local proposal number (usually a non-negative number come up by each proposer) and node ID (e.g. machine or process ID) as the proposal number we use in our protocol message. The advantage of having node ID included in the proposal number is that even if both proposers come up with the same local proposal numbers we can still break ties between proposals sent by those machines. (We suppose there are no two machines using the same ID.) Notably, default derived `Ord` typeclass will compare the local proposal number before comparing the node ID, which is exactly what we want for this purpose.

```haskell
data ProposalId nodeId = ProposalId Word64 nodeId
  deriving (Show, Eq, Ord, Typeable)
```

For each acceptor, it should keep the highest proposal number along with the value it has already accepted. This piece of information will be embedded in *promise* messages during the first phase when acceptors respond to proposers' *propose* message.

```
-- | Representation of something accepted by an acceptor
data AcceptedValue nodeId value = AcceptedValue {
                                acceptedProposalId :: ProposalId nodeId,
                                acceptedValue :: value
                                } deriving (Show, Typeable)
```

For each different message type used in the two-phase algorithm, we define a new type and have a union type wrap around all those different message types. Since it is possible that an acceptor has not accepted any proposals when responding to a promise, we use a `Maybe` type to embed `AcceptedValue` in `Accept` message.

```
-- | Union type for all types of messages
data Message nodeId value = MsgPrepare (Prepare nodeId)
                          | MsgPromise (Promise nodeId value)
                          | MsgAccept (Accept nodeId value)
                          | MsgAccepted (Accepted nodeId value)
                          deriving (Show, Eq, Typeable)

{- Prepare message from proposers in first phase -}
data Prepare nodeId = Prepare (ProposalId nodeId)
  deriving (Show, Eq, Typeable)

{- Promise message from acceptors in first phase -}
data Promise nodeId value = Promise (ProposalId nodeId)
                                    (Maybe (AcceptedValue nodeId value))
  deriving (Show, Eq, Typeable)

{- Accept message from proposers in second phase -}
data Accept nodeId value = Accept (ProposalId nodeId) value
  deriving (Show, Eq, Typeable)

{- Accept acknowledgement from acceptors in second phase -}
data Accepted nodeId value = Accepted (ProposalId nodeId) value
  deriving (Show, Eq, Typeable)
```

This definition of `Message` type is not satisfactory now. The additional wrapper type incurs some redundant boilerplate pattern matching code when receiving a message. We will leave it for future improvement. We also define an `Action` type in our implementation. This type specifies all the impure side-effects required by the protocol itself. For Paxos, the side effects are mainly sending messages to other machines or processes. To reduce the number of actions, we introduce an action (or event) named `Broadcast`, which can used in the first steps of both phases on the proposer side.

```
-- | Broadcast group identifier
data BroadcastGroup = Acceptors | Learners deriving (Show, Eq)

-- | Actions which might need to be executed as the result of a state
-- transition
data Action nodeId value = Send nodeId (Message nodeId value)
                         | Broadcast BroadcastGroup (Message nodeId value)
                         deriving (Show, Eq)
```

For each role in the Paxos algorithm, we implement a module that includes the operations needed for that particular role in the algorithm. A simplified design interface is as follows. The style of design is largely affected by object-oriented programming, because the code is more modular and cleaner that way.

```haskell
module Proposer (initialize, sendProposal, handlePromise, nextProposalId)
module Acceptor (initialize, handlePrepare, handleAccept)
module Learner (initialize, handleAccepted, getValue)
```

3.1 Customization of Paxos Algorithm

The learner role is used to learn whether a value has been chosen by the majority of machines or processes in the group. [2] achieves this by having each acceptor, whenever it accepts a proposal, respond to all learners, sending them the accepted proposal. A cost-effective way is recommended that every acceptor should send accepted value to one leader learner, and let the leader learner propagate the learned value to other learners in the group. In our implementation, we simply make every acceptor send accepted proposals to every learner in the system. As soon as one learner gets the majority votes from the group, we can guarantee that a consensus has been reached in our system. As discussed in [2], this approach is expensive in that it incurs the cost of one-to-all communications. On the other hand, it can quickly terminate one round of algorithm when there are a lot of packet losses and delays in the network.
Also, we resolve the unsuccessful attempts of proposal phase by using random waiting. Although it is a less efficient approach than some other Paxos variants (e.g. Raft), it is simple and easy to implement.

3.2 Pure Message Handlers

By introducing `Action` types into our implementation, we can delay the side effects of the protocol and have pure message handlers in Haskell (see below). The message handlers just implement the core state transitions in the algorithm and the side effects (i.e. `Action`s) can be carried out by impure Haskell code (i.e. `IO` operations). This separation of concern helps us to debug the code and organize the logic. Notice that these message handlers are also stateless, where each call can finish without consulting any internal states of each machine or process. It is conformed to when programming in Haskell.

```haskell
-- | Handle a single prepare message received from a Proposer
-- | handlePrepare :: old state -> rcvd message -> new state -> delayed side effects
handlePrepare :: State -> Message -> State -> [Action]
```

4 Experience

We tested our implementation of Paxos algorithm locally. To simulate an unreliable network environment, we write our own unreliable message passing interfaces with random packet drops and delay. The unreliable write interface is as follows.

```haskell
-- | Unreliable write a value to a TChan
writeChan :: Double -> (Int, Int) -> Chan a -> a -> IO ()
writeChan lost delayBounds chan msg = do
  rndDrop <- randomIO
  rndDelay <- randomRIO delayBounds
  unless (rndDrop < lost) $ void $ forkIO $ do
    threadDelay rndDelay
    atomically $ TChan.writeTChan chan msg
```

We tested our implementation on a topology of 2 proposers, 5 acceptors and 2 learners with a simulated packet drop rate of 2% and 20% and a network delay of $1000 - 300000$ milliseconds. By design, the value proposed by proposers with higher proposal numbers will always win at the end of a round. When the packet loss is increased to 20%, it will take significantly longer for the group to reach consensus.

5 Conclusion

In this project, we implemented the basic Paxos algorithm in Haskell. The total lines of code is around 600, which is considerably smaller than implementations in traditional languages. During the process, we applied some of the useful Haskell features learned in class. In future, we plan to extend the project to some distributed services that can be built upon the basic Paxos algorithm.

References

[1] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[2] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[3] David Mazieres. Paxos made practical, 2007.

[4] Roberto De Prisco, Butler Lampson, and Nancy Lynch. Revisiting the paxos algorithm. *Theoretical Computer Science*, 243(1):35–91, 2000.