Distributed Systems

Paxos synod simulator

Utah Tech University—Department of Computing

Spring 2024

Overview

This project is a simulator for the Synod protocol of Paxos, i.e., a single decision as described in *Paxos Made Simple*.

You will write a simulator that implements the protocol, with a few key actions being controlled by a script. This will allow you to simulate various interesting scenarios in a predictable and repeatable way.

First steps

- Download the project using CodeGrinder. You will essentially be starting from scratch. This will be a single-threaded project with no network component. Instead, you will be implementing a state machine.
- Think about the state you need to track to simulate a cluster of Paxos nodes. There are a few basic kinds of state you will need to track:
  1. The state of each node. You will be simulating everything in a single process, but the state that each node keeps track of should be kept isolated from other nodes so the simulation will be realistic. You will not know how many nodes to track until the input script tells you, so wait until that message arrives to initialize the nodes.
  2. The list of messages in transit on the network. The script will control the sequence of events mainly by controlling when a network message is delivered. This allows us to simulate slow networks, fast networks, out-of-order deliver, duplicate delivery, and message loss. We can also simulate a crashed node by not delivering any messages to it.
- A simple way to structure this is to bundle all of the state into a `struct` type called `State` and then write most of the simulation code as methods on that object.

Main loop

Since all actions are triggered by a line of input, the main loop of the simulator just reads lines and invokes methods to handle them. You can start with something like this.

```
scanner := bufio.NewScanner(os.Stdin)
for scanner.Scan() {
    line := scanner.Text()

    // trim comments
    if i := strings.Index(line, "//"); i >= 0 {
        line = line[:i]
    }

    // ignore empty/comment-only lines
    if len(strings.TrimSpace(line)) == 0 {
        continue
    }
    line += "\n"
    switch {
    case state.TryInitialize(line):
    case state.TrySendPrepare(line):
    case state.TryDeliverPrepareRequest(line):
    case state.TryDeliverPrepareResponse(line):
    case state.TryDeliverAcceptRequest(line):
    case state.TryDeliverAcceptResponse(line):
    case state.TryDeliverDecideRequest(line):
    default:
        log.Fatalf("unknown line: %s", line)
    }
}
if err := scanner.Err(); err != nil {
    log.Fatalf("scanner failure: %v", err)
}
```

Main loop

- A `switch` statement with no value evaluates each case and stops when one of them evaluates to true. In this code, each `state.TryXYZ` method returns `true` if it recognized and processed a command, and `false` otherwise.
- The `state` object tracks the state of the nodes and all of the network messages. The simulator updates `state` as it responds to each event.
- Each handler parses a message using code like this:

```
var size int
n, err := fmt.Sscanf(line, "initialize %d nodes\n", &size)
if err != nil || n != 1 {
    return false
}
```

- The precise state maintained by each node and how network messages are stored will probably change as your proceed. Start with something simple and evolve your design as you go. Do not try too hard to predict where you will end up: if you are writing the handler for a message and realize that you need information that you are not tracking, go add it to the appropriate objects and continue working. It is not realistic to anticipate everything in advance.

Network simulation

To find and deliver a message from the network, you will need to identify each message using a key like this one:

```
const (
    MsgPrepareRequest = iota
    MsgPrepareResponse
    MsgAcceptRequest
    MsgAcceptResponse
    MsgDecideRequest
)

type Key struct {
    Type   int
    Time   int
    Target int
}
```

Each message can be uniquely identified by the message type, the time it was sent, and the node it was sent to. Each delivery command in the script will include all three of these identifying elements when instructing the simulator to deliver a message.

In your `state` object, store a `map` from `Key`s to the message contents. Do not delete an item from the network when it is delivered, because sometimes a network delivers a message multiple times.

Initialize

The initialize message takes the form:

```
initialize 3 nodes
```

Anywhere between 3 and 9 notes is allowed, and you should calculate how many nodes is a majority in each case. The initialize message will always be the first message and will never be repeated.

When this message arrives, initialize the set of nodes in your state object. They can start out mostly empty and you can fill them in as you program the rest of the simulation.

Have the `TryInitialize` method return `true` if it recognizes the message and `false` if it does not parse correctly. In the latter case the main loop will continue trying.

Send prepare

The next command to process is of the form:

```
at 1001 send prepare request from 3
```

In this example, 1001 is the timestamp when node 3 should act as a proposer and initiate a prepare round by sending a prepare request to each of the nodes.

Send a message by adding it to the network and print each message as you send it. You can decide how to store the content of the message.

Note: this is the only part of a proposer that is controlled directly by the script. The proposer should maintain enough state that it can react appropriately to the responses it receives from the other nodes. In particular:

- When it receives a majority of positive propose responses or a majority of negative propose responses it should move on to the accept round (for a positive result) or abandon the attempt and start a new prepare round (for a negative result).
- When it receives a majority of positive/negative responses from the accept round, it should either move on to the decide round (for a positive result) by sending out decide messages, or it should abandon the attempt and start over with a new prepare round (for a negative result).
- Each node should start proposals using `5000 + node` (where nodes are numbered starting from 1) and each time it has to restart it should add 10 to its proposal number.

Proposer

The proposer role is spread out. The 25-line Paxos summary has this to say about the proposer:

```
1  proposer(v):
2    while not decided:
3      choose n, unique and higher than any n seen so far
4      send prepare(n) to all servers including self
5      if prepare_ok(n, na, va) from majority:
6        v' = va with highest na; choose own v otherwise
7        send accept(n, v') to all
8        if accept_ok(n) from majority:
9          send decided(v') to all
```

But your code will not run in a loop like this. Instead, you must maintain enough state on the node so that when a response message is delivered you can react as though the pseudo-code above was running.

Note: the pseudo-code says to choose *n* higher than any *n* seen before. We will use a simpler approach: each time the proposer must start over, add 10 to the value of *n* it used previously.

Deliver prepare request

The next message to handle is of the form:

```
at 1002 deliver prepare request message to 2 from time 1001
```

This asks a node to respond to a prepare request sent at time 1001 and delivered to node 2. Any actions triggered by this message (a prepare response in this case) should happen at timestamp 1002.

The 25-line Paxos has this to say about the acceptor's prepare handler:

```
10  acceptor state on each node (persistent):
11   np     --- highest prepare seen
12   na, va --- highest accept seen

13  acceptor's prepare(n) handler:
14   if n > np
15     np = n
16     reply prepare_ok(n, na, va)
17   else
18     reply prepare_reject
```

It should send a prepare response message back to the sender based on its node state. You may need to add state to your node at this point and initialize it correctly if you have not already.

Deliver prepare response

The next message takes the form:

```
at 1005 deliver prepare response message to 3 from time 1002
```

Note that a prepare response can be a positive (okay) or a negative (reject) response. The response is sent in reply to a prepare request, so the script assumes that it is sent at the timestamp when the request was delivered.

A prepare response will cause one of a few potential actions:

1. Record the vote and do nothing else (not enough votes back yet)
2. Record the vote, note a majority of positive votes has just been achieved, and move on to the accept round by determining the correct value to nominate and sending accept requests to all nodes.
3. Record the vote, note a majority of negative votes has just been achieved, so give up on this proposal, add 10 to the proposal number, and start over with a fresh prepare round.
4. Note that this round was already resolved before this vote was received, so ignore it.

Watch out for duplicate messages, delayed messages from previous attempts, etc. Record whatever state you need to so that the appropriate action can be taken immediately when each message is received.

When the proposer gets to pick its own value, use `11111 × node` so the value will be easy to associate with the node number.

Deliver accept request

The next message is sent by a proposer after a successful prepare round and takes the form:

at 1009 deliver accept request message to 1 from time 1006

The 25-line pseudo-code lays out the behavior:

```
19  acceptor's accept(n, v) handler:
20   if n >= np
21     na = n
22     va = v
23     reply accept_ok(n)
24   else
25     reply accept_reject
```

As before, verify that you are tracking enough state to be able to response appropriately to this request.

Deliver accept response

Next we have:

```
at 1011 deliver accept response message to 3 from time 1008
```

This is another message for the proposer, so as before it must decide whether to take immediate action (succeed and move on to the decide round, fail and go back to a new prepare round), record the vote while waiting for more responses, or ignore it because it is a duplicate, the round has already resolved, or it is a delayed message from a prior round.

If the accept round is successful, the proposer should send out decide requests to all nodes. There is no response to a decide message, so the proposer's responsibilities are concluded once it sends out the decide requests.

Deliver decide request

The final script instruction has the form:

```
at 1014 deliver decide request message to 1 from time 1012
```

The receiver records the decision and can now act on it. In a replicated database, this would be the correct time to apply the command to the local database replica. A message to the console should suffice here.

Note that if you have already recorded a decision, it should be impossible to receive a conflicting decision if everything is implemented correctly. Check for this and complain if you detect a problem. It is perfectly fine to recieve the same decision multiple times.

Example script with single proposer and no failures

```
initialize 3 nodes

// node 3 starts a proposal
at 1001 send prepare request from 3

// deliver all three prepare requests, each triggering a response
at 1002 deliver prepare request message to 2 from time 1001
at 1003 deliver prepare request message to 3 from time 1001
at 1004 deliver prepare request message to 1 from time 1001

// deliver the first prepare response
at 1005 deliver prepare response message to 3 from time 1002

// two responses is a majority so the second will trigger accept requests
at 1006 deliver prepare response message to 3 from time 1003

// deliver the third response, which will not affect the outcome
at 1007 deliver prepare response message to 3 from time 1004

// deliver all three accept requests, each triggering a reponse
at 1008 deliver accept request message to 3 from time 1006
```

Example continue

```
// at the moment the second acceptor votes in favor, consensus has been achieved
at 1009 deliver accept request message to 1 from time 1006
at 1010 deliver accept request message to 2 from time 1006

// deliver the first accept response
at 1011 deliver accept response message to 3 from time 1008

// at the moment the second acceptor vote is counted,
// consensus has been discovered by the proposer
// and decide requests will be sent out
at 1012 deliver accept response message to 3 from time 1009

// deliver the third response, which will not affect the outcome
at 1013 deliver accept response message to 3 from time 1010

// deliver all three decide requests
// nodes can act on the decision when they receive these requests
at 1014 deliver decide request message to 1 from time 1012
at 1015 deliver decide request message to 2 from time 1012
at 1016 deliver decide request message to 3 from time 1012
```

Example solution

You can download a pre-compiled implementation to play with using:

```
curl -so synod-example https://computing.utahtech.edu/cs/3410/synod-example
chmod 755 synod-example
```

and launch it using:

```
./synod-example
```