Jake Sanders
Hampton Terry
CSE 452 Project 3+4 writeup

(we didn't have a linux machine handy to .tar up our code. so apologies for the .zip)

We have added a new class in addition to TwitterNode: PaxosModule. PaxosModule maintains all of the state associated with the paxos algorithm, and produces the right responses to prepare/promise/accept/accepted/learn requests.

We've isolated the code associated with paxos as much as possible, with the bottom portion of TwitterNode being dedicated with paxos-node-specific commands.

Additionally, we've migrated over to using json maps for storing data in files/sending data over the network. Hopefully this should aid in your debugging (since you can read the contents of packets and files easily).

We have 3 distinct classes of nodes in our implementation: **client**, **server**, and **paxos** nodes. Each paxos node simultaneously acts as a proposer, acceptor, and learner. Paxos nodes store a list of rounds and their associated values. Server nodes store the actual file contents, and act to process the actions of transactions decided on by the paxos nodes.

- These roles are assigned by the "**n assign (role)**" command.
- Client n is told to initially use server m by the command "**n useServer m**".
- Server n is told to initially communicate with paxos node m using "**n usePaxosGroup m**".
- Paxos node n is told to join m's existing paxos group using "**n joinPaxosGroup m**".
- The initial membership of a paxos group that n should be a part of is defined using the "**n startPaxosGroup n i j k …**" where n is required in the membership. This must be called first to initially seed the paxos group before other nodes can join using joinPaxosGroup.

The normal control flow proceeds like so:

- A client starts a transaction, performs a series of actions, and commits
- Upon receiving a new action from a client (including a commit), the server compares the 'proposed round' for all of the actions in the transaction (including reads) with the timestamps that are on disk already. If the file timestamp is greater than or equal to the proposed round for the transaction, we know that the transaction has read/wrote stale data and must be aborted. Otherwise, it submits to paxos the transaction (i.e. value) for consideration on that round (the transaction data includes the requestId, clientId, transactionId, and list of actions performed on that txn.
- The paxos node receives the request and either
    - Starts voting on that round, if it doesn't think one has started for it, by sending prepare requests to all of the nodes in paxos
    - Ignores the request, if a round of voting has started but not finished
    - Replies with the value learned for that round, if the round has finished

- The node then performs the paxos protocol for reaching consensus on that round, using the other nodes that it knows about.
- When a majority have accepted, it sends the value to all the paxos nodes to learn.
- When a paxos node learns, it stores that result and sends an update containing that value to all the servers that it knows about (this last part is not strictly necessary, a tardy server will be updated when it makes another request to paxos).
- When a server receives an update, it either
- Performs the writes associated with that transaction, remembers that that transaction has been completed, and send a commit acknowledgment to the client that made the commit request.
- Ignores the update if is old, or rejects the update and sends a catch up request to all the paxos nodes it knows about if the update is to a future round.
- The client notes the commit acknowledgment, and moves on to the next request

The way we ensure serializability is by using a notion of **rounds** which describe the sequence number that a transaction should be executed on. Starting at 0, servers submit transactions to paxos corresponding to a particular round. The way we ensure that transactions are not processed by servers out of order is by rejecting any transaction submissions that are not for the round that we currently think it is on the server. When this happens, the server sends a request to all paxos nodes that it knows about, saying that it needs to recover from a given round number (where the round number is the current **round** on that server). The paxos node will reply with all of the learned values starting at the request round. The server only increments its **round** when it receives an update for the current one. Servers use the **round** as the file version, which means that it can detect if it was requested to reapply an old update, and ignore it.

Clients are initially seeded with one or more servers that it should communicate with, but add to this list as more servers communicate with it (e.g. when a commit confirmation is received). Clients make request on the lowest number server that they know about, and retry requests 5 times before marking a server as dead (which can be reversed) and moving that server from the list of good servers to the list of bad servers. If a client is about to invalidate the last of its good servers, it moves all of the bad servers back to the good list and the process starts from the beginning.

We initialize the current **round** to 0 whenever a paxos or server node first initializes. This has many useful purposes. This means that the first request that any node makes will always conflict with the very first action undertaken by the system, which greatly simplifies startup/catchup (i.e. if a node is behind, it can send a request to paxos for all the updates starting at that nodes current **round**). This allows a paxos node to simply return the value learned for a given round if a request is received relating to that round. This also means that a given machine cannot leapfrog over updates, since the **round** that a given machine believes it to be is exactly 1 more than the last update that machine has received. This is pivotal in our implementation of

project 4, since this essentially blocks new paxos/server nodes from harming the state of the system after they initially start up, while providing a convenient mechanism to automatically detect tardiness.
In order to bootstrap a paxos group, call “n startPaxosGroup a b c” on a node after it assigning it the paxos role. This explicitly sets the nodes that should be considered in the paxos group. At the very least, you should use “n startPaxosGroup n” to bootstrap the creation of the initial group. From there, you can use “m joinPaxosGroup n” to have additional nodes join n’s group.
Once you have a paxos group set up, you can use the “**n paxosLearn (value) (round)**” command to have the specified node act as a proposer, accepting your value/round request as it would a server’s, and running the paxos protocol to attempt to reach consensus.
You can use the “**clingToLife**” command on a node in order to get it to start a callback with spins until you call “**embraceDeath**” on that same node. This extends the life of the simulation until a point of your choosing.

You will probably need to eliminate all of the saved state on disk between executions if you wish to get a clear picture. On the plus side, the contents of all of the files are stored as plain text (json), so they’re a convenient way to know the state of a file/paxos instantiation.

Project 4:

In order to implement the ability to permanently add/remove nodes from the system, we designed our project around the idea of a **round** which starts at 0 that allows for a server/paxos request to be ignored if the particular node is behind, and for that round to become aware that it needs to catch up. This makes server addition/recovery trivial, as any request it makes to paxos will be met with a response for an update.

What was less trivial was updating paxos group membership. We accomplished this using paxos itself to decide on and propagate membership changes while dealing with the possibility of simultaneous server requests. By proposing a membership update as the value for a given round, and by adding code to detect membership updates on the paxos/server nodes, we can ensure that the state of the paxos membership does not change mid-round, and it also offers us a convenient propagation mechanism. When a paxos node receives a learn request on a membership update, it compares the groupVersion of the proposed update to the current one, and updates the membership if the proposed group is newer (in order to account for duplicate/late queries). When a server receives a membership update, it will immediately updates the paxos membership, since updates are ignored if they have not been applied in sequence.

We also have a “**n removeFromPaxos m”** command which instructs paxos node n to propose a new group membership which excludes m. This should be used after m has been taken offline.

The included “proj3script” cycles through bootstrapping a paxos group, having 2 additional nodes join the group, and finally eliminating one of those nodes from the group.

After nearly 40 hours of work myself on these two projects, chasing inscrutable bugs, and with other deadlines looming, I know when I need to call it quits. This final stage has defeated me. I wish that either the labs had been assigned in reverse order (to be bottom-up instead of top-down), or that we has spent a lot more time designing a truly maintainable codebase (probably both). Either way, integrating Paxos into our existing code has proved to be an insurmountable task. We can only hope that the scripting hooks I have given you guys to prove that the paxos implementation and the code to join nodes to/ remove them from an existing paxos group works well, and that you will bestow your mercy upon us.

Thank you guys for a brutal, interesting quarter.

Summary of simulator commands:

- **assign:** “n assign (role)” assigns a role to a node. This should be performed after the node is started and before other commands are made to it. Choices are **client, server,** and **paxos**.

Commands to a paxos node:

- **startPaxosGroup**: “n startPaxosGroup n i j k” tells n to initialize paxos membership to include nodes n, i, j, and k. This needs to be performed on at least 1 node in order to start a new paxos group, and should always include n where n is the node being commanded. Other nodes should then be able to use...
- **joinPaxosGroup:** “n joinPaxosGroup m” instructs n to attempt to send a join request to

m, which generates a change group membership update. This update is done using paxos, so n may not effectivley be a member of the group for a while. The node continually retries until the request is granted.

- **removeFromPaxos:** “n removeFromPaxos m” instructs paxos node n to start a group membership update request that excludes m. The node continually retries until the membership is updated.
- **paxosLearn**: “n paxosLearn (value) (round)” instructs paxos node n to act as a proposer, processing the request in the same manner as it would a server’s. This is useful for isolating our paxos implementation from our server implementation (which doesn’t work so well).

Commands to a server:

- **usePaxosGroup**: “n usePaxosGroup m” instructs server node m to include m in its list of paxos nodes to submit requests to.

Commands to a client node:

- **useServer:** “n useServer m” tells client node
- **create**: “n create username” tells n to submit an account creation request
- **login:** “n login username” tells n to submit a login request on username
- **logout:** “n logout” tells n to logout
- **post**: “n post blah blah blah blah” tells node n to submit a new tween with contents “blah blah blah blah”
- **add:** “n add user2” tells n to add user2 to the currently logged in user’s following list
- **delete:** “n delete user2” tells n to unfollow user2 on the currently logged in user’s account
- **read:** “n read” tells n to request all of the unread tweets from users that the current user is following