Liangbin Zhu and Brendan Cross
Distributed System and Algorithm – CSCI 6510

Distributed Flight Reservation System

Paxos algorithm

Part I - File Structure & Data Structure

Our application use Java and the main program consists of five parts:

Service.java

- Main entrance for the application. Initialize the log and reservation data structure and handle site recovery.
- Start acceptor and learner threads to keep receiving incoming message. (Acceptor.java, Learner.java)
- Starts main thread for handling local command. (CmdHandler.java)

CmdHandler.java

- Deals with all the user input from command line.
- Detects and fill holes before executing “reserve” and “cancel” operations.
- Starts Synod instance to determine new log entry.

Proposer.java

- Communicates with all the acceptors using UDP socket.
- Serves as a distinguished learner to send commit messages.
- Be responsible for query maximum log entry number during site recovery.

Acceptor.java

- Keeps receiving messages from proposers using UDP socket.
- Maintains the acceptor state variables for different log entry, and is responsible for saving the state into stable storage.
- Updates reservations and log data structure, then saves new log entry (before next checkpoint) into stable storage.
- Communicates with learner thread when the new log entry reaches or exceeds a checkpoint, using blocking queue.

Learner.java

- Keeps receiving notification from acceptor through blocking queue.
- Detects and fills holes when the log entry reaches or exceeds a checkpoint.
- Save reservations and new log entry (checkpoint or larger log entry) into stable storage.

The data structure consists of two parts:

Reservations

- A list of reservation objects, containing the client name and flight numbers.
- The reservations data structure keeps the most recent information

Reservations (Backup)

- An extra list of reservation, containing the same content as the reservations data structure.
- The reservation backup data structure only keeps the reservation status before the next checkpoint. It is used to save the reservation information when reaches checkpoint.

Log

- A TreeMap data structure, whose key is the log entry number and value is log record object.
- The log record object contains event type (reserve or cancel), reservation information, checkpoint mark and the site name that made the reservation.

Part II - Implementation

1. Multi-thread

The program uses three threads in total, including main thread, acceptor thread and learner thread.

In the main thread, it deals with site recovery and receives user’s input from terminal. When it receives “reserve” or “cancel” command, it would first detect and fill holes in the log, then starts a new Synod instance to decide the value. After the execution of the Synod instance, it would print the result of this execution to the user.

In the acceptor thread, it is responsible for handling messages from proposers, that is, replying the corresponding “promise”, “ack” and “nack” message to proposer. Moreover, it also receives the final “commit” messages from proposer (distinguished learner), and then updates the log and reservations data structure. When it receives a log entry number that is equal to or larger than the next checkpoint, the acceptor would notify the learner thread to start handling checkpoint.

In the learner thread, it is responsible for detecting holes before the checkpoint and save the reservation backup and log into stable storage. The learner thread shares the same blocking queue with acceptor thread for communication.

2. Stable storage

Every time when the acceptor updates its state variables, it would save the information into the stable storage. If the acceptor receives a commit message that is new to the site, it would add this new log entry in the end of log file finally. When the log entry reaches a checkpoint, it would also save the whole reservation backup data structure in the stable storage.

3. UDP Sockets

Each site uses two UDP sockets. The first one is for the proposer. It uses the first port in the given port range to communicate with all the other acceptors (including this site itself). The other one is for the acceptor. It uses the last port in the given range to communicate with proposers (including this site itself).

Part III - Project Details

1. Learner Implementation

When the acceptor thread receives a “commit” message, it would first check whether the log number is already in our log. If the log entry is already learnt, then the acceptor would ignore this message. Otherwise, it would save the new log entry in the log in memory and then compute the next checkpoint location based on the most recent checkpoint in the log list.

If the current log number is less than the next checkpoint number, it would update the both reservations and reservation backup data structures, and then append this new log at the end of log file.

If the current log number is equal to the next checkpoint number, it would also update the both reservations and reservation backup data structures, but did not add this new entry to the log file at that time. Then it would send a notification to learner thread to start handling checkpoint.

If the current log number is larger than the next checkpoint number, it would only update the reservations data structure. Then it sends a notification to the learner thread to start handling the missed checkpoint.

2. Recovery Algorithm

When site recovers from crash, it would first send “query” messages to all the other site to ask about the maximum log entry number known so far. It would wait for the “reply” messages within a timeout, then choose a maximum number it received as the most recent log entry number. Then the site would execute multiple Synod instances one by one to learn the value for the missed log entry until the max log entry is learnt. In the recovery process, when the acceptor receives a new “commit” message it would not update the reservations data structure, but only save the log entry into log. After filling these holes in the log, the program would replay all the logs after the most recent checkpoint to recover the reservations data structure. Every time when it reaches a checkpoint during replaying log, it would save the reservations backup and mark a checkpoint in the log file.

The recovery algorithm fills holes in the log through executing multiple Synod instances, so it could guarantee the safety and liveness. If the site cannot receive any reply message when recovery, it will also be able to learn these missing holes when make new reservations.

3. Checkpoint Strategy

The program uses the learner thread to handle checkpoint operations. The learner thread keeps waiting message from acceptor. Every time when the learn receive a new message, it would compare the log number with the next checkpoint number.

If the current log number is smaller than the next checkpoint number, that means the previous checkpoint has already by handled, then the learner updates the reservations backup data structure and append this new log entry at the end of log file.

If the current log number is equal to the next checkpoint number, that means it reaches a check point. In that case, the learner would detect and fill all the holes before this log number, then save the reservations backup data structure in the stable storage, and append the log entry at the end of log file.

If the current log number is larger than the next checkpoint number, that means it missed a previous checkpoint. In that case, the learner would detect and fill all the holes before that checkpoint, then save the reservations backup data structure in the stable storage, and append the checkpoint entry at end of log file. Then it would put the current entry back to the blocking queue to check whether there is other checkpoint missed.

Part IV - Test Case

We created a series of test cases to verify that the application was performing as expected. These tests cases can be broken down into several categories:

- User Input validation:
  - Basic test to ensure user commands performed their designed actions.
- Communication Tests:
  - These basic tests just ensure that when we send a message to everyone that everyone receives what was intended. Involved making reservations at a site and observing the behavior in it and other sites.
- Paxos Optimization Tests:
  - These tests confirmed the optimization was working, being applied to the proper site, and that it would not somehow give the proposer advantage in the wrong log entry.
- Checkpoint Tests:
  - These tests were created to verify that checkpointing was working as intended. We tested to confirm the checkpoints happened on time, that they would survive recovery and that they would still operate properly under frequent failure.
- Recovery Tests:
  - These test cases verify that the site's state was recovered when the site came back online. These tests were closely tied to the checkpoint tests, but were more focused on the filling of holes in the log and the maintaining relevant role information for our site.
- Flight Reservation Tests:
  - These last tests were just a variety of scenarios to confirm we could not cancel a flight reservation that was already cancelled or that we couldn't overbook, etc... This was mainly a test that the flight reservation system that is using Paxos is working properly.

The tests were run manually on a local docker setup. Dropped messages were simulated through killing the sites that we didn't want receiving a given message. We ran the tests on 3 – 5 docker instances.

Part V - Team Work Division

Liangbin Zhu: acceptor code, checkpoint implementation, learner code, hole detection
Brendan Cross: proposer code, Paxos optimization, site recovery, application testing