Project 2 - Distributed Flight Reservation System - Paxos Version
CSCI-4510/6510 – Fall 2019

Assigned: Oct. 29, 2019
Teams formed by: Nov. 5, 2019 (10% penalty for missing this deadline).
   Work in groups of two or three. Your entire team must be enrolled in the same course number.
Docker Package Notification Deadline: Nov. 15, 2019
Project due: Tuesday, Nov. 26, 2019 at 11:00pm
Demos: Dec. 2 – Dec. 6, 2019

1. Summary

In Project 1, you implemented a distributed flight reservation service using the Wuu-Bernstein replicated log and dictionary algorithm. This algorithm has the benefits of simplicity and low message overhead, but it also allowed conflicting flight reservations to be made. Thus, you needed to implement a conflict resolution algorithm to automatically cancel conflicting reservations.

In this project, you will implement a distributed flight reservation service using the Paxos Algorithm to maintain a replicated log of “reserve” and “cancel” events.

The system consists of N sites. Each site plays the role of a Proposer, issuing proposals for “reserve” and “cancel” operations. It also plays the role of an Acceptor, helping to determine which proposal should be accepted for each log position. And, it plays the role of a Learner; it stores a replica of the log in stable storage and updates this log with committed log entries. The Learner can also store a data structure that contains the current reservations, but this data structure should not be stored in stable storage (see below).

The log replicas will be maintained in a way so that it is not possible to create conflicting reservations.

2. Implementation Details

When a user tries to make a reservation at a site, the site’s Proposer should execute the following steps:

1. If there are any holes in the site’s log, execute the Synod algorithm for those log entries to learn the missing values. Update the site’s reservation data structure with any newly learned reservation information.
2. Check whether the user’s desired reservation conflicts with any reservations in the updated reservation data structure. If so, do not make the reservation. If there is no conflict, execute the Paxos algorithm as the Proposer to propose the reservation for the first empty log entry.

To cancel a reservation, the site’s Proposer performs the same two steps. The Proposer should not issue a proposal to cancel a reservation if the reservation was already cancelled previously. In this case, the “cancel” request should fail.

In the case of timeouts when waiting for “promise” or “ack” messages, the Proposer should retry the proposal a maximum of two times (three tries total) per log position. The proposer should increment its proposal number in between each try. Even if there is no timeout, due to concurrent proposals, the
user’s reservation may not be committed to the log. If the user’s operation is not added to the log, the application should notify the user according to the UI specification.

Note 1: You will need to use multiple threads or asynchronous IO for proposals. A Proposer should not try to contact the Acceptors one at a time, but rather it should send messages to all Acceptors at once and wait for responses from a majority.

Note 2: Your site should be able to act as a Proposer, an Acceptor, and Learner simultaneously.

Learning Committed Values (all groups): You can use either of the two strategies discussed in class to inform the learners of committed values. If you use a “Distinguished Learner”, the site that is the Proposer for the committed value can also be the “Distinguished Learner”.

Stable Storage (all groups): The log should be stored in stable storage. The reservation data structure should not. Your application should also store the Acceptor state variables in stable storage as specified in the Synod Algorithm.

Site Failures and Recovery (all groups): Your application should follow the Paxos algorithm, meaning it should tolerate message loss and site failures. The application should be able to create new log entries so long as a majority of sites are active.

When a site is failed, it may miss commit messages for some log entries. When the site recovers, it should learn the values for these missed log entries. You must design and implement a recovery algorithm to do this. Your algorithm should ensure the safety and liveness guarantees described in the Paxos paper.

Checkpointing (CSCI 6510 only): When a site recovers, it must recreate its in-memory copy of the reservation data structure by replaying the log (executing the operations in order). This can be time-consuming if the log is large. To speed up recovery, checkpointing should be used. A site should save a copy of the reservation data structure to stable storage after every 5 log entries are committed (in a real-world implementation, checkpointing would be done after many more events). The site should fill in any holes in the log before saving the checkpoint.

Note: When a site learns a committed value for log entry 5, for example, it should fill in any holes in log entries 1 - 4, and then it should create a checkpoint of its reservation data structure, which has the state corresponding to log entries 1-5. If it misses log entry 5, when it creates a log entry with number larger than 5, it should fill in any holes and then create the checkpoint for log entries 1 - 5.

When a site recovers, it should load the most recent checkpoint of the reservation data structure into memory. After the site learns any missed log entries (using your recovery algorithm), the site should replay the log, starting from the first event after the most recent checkpoint. This will bring the in-memory data structure up to date. (Note that new log entries may be created during a site’s recovery, so the site may still not know about the most recent log entries. It will learn them when it issues a proposal.)
Paxos Optimization (groups of 2): Your application does not need to use a Distinguished Proposer, so you do not need leader election. You will still implement a small optimization to the Paxos algorithm. When a site “wins” a log position k, meaning the site’s proposed value is the one that is committed to log position k, this site can skip the “prepare-promise” phase for its first proposal for log entry k+1 (if it has a proposal for this log entry). In essence, this site’s Proposer has implicit promises from all Acceptors for its first proposal for log entry k+1. This first proposal should have proposal number 0. Note that Acceptors should check that the proposal that skips “prepare-promise” phase has permission to do so.

Full Paxos (groups of 3): Your application should use a leader election algorithm of your choice to elect a Distinguished Proposer. This leader election algorithm should detect the failure of a current leader and elect a new one. It should also allow recovered processes to learn the identity of the leader (and possibly elect a new leader on its recovery). The Distinguished Proposer should use the optimization for skipping the “prepare-promise” phase as described in the Paxos paper. Should the leader election algorithm elect more than one leader, the Distinguished Proposer will be the first of these leaders to “win” a log position, meaning that site’s value is committed to the log. Every site will send its proposed operations to the Distributed Proposer, and this site will execute the steps at the beginning of Section 2 for each proposed operation, in the order that it was received at the Distinguished Proposer.

3. UI Specification

(a) To make a reservation, the user enters the following:

% reserve <client_name> <CSV_list_of_flight_numbers>

If there are no conflicts in the local copy of the reservation list, the system should submit the reservation (see below) and print out:

% Reservation submitted for <client_name>.

Otherwise, the application should print:

% Cannot schedule reservation for <client_name>.

(b) A client can cancel their flight reservation by entering:

% cancel <client_name>

After the system cancels the reservation, it should print:

% Reservation for <client_name> cancelled.

If the reservation cannot be cancelled, the system should print:

% Cannot cancel reservation for <client_name>.

(c) The system should display the list of all reservations using the following command:

% view
In response to this command, the system should list all reservations in the site’s copy of the reservation list, sorted in lexicographical order by *client_name*, in the following format:

- userA 1,3
- userB 1
- userC 2,4

(d) The system should also support the following command to display the contents of the log:

% log

In response to this command, the system should list the log contents, sorted in ascending order of log position:

- reserve userA 1,3
- reserve userB 1
- cancel userA

(e) **Messages:** we would like to observe which messages are sent and received by your application. To allow this, please print a message to stderr every time a site sends or receives a message. The output should look something like (this will not be checked by the autograder):

% sending prepare(100) to all sites
% received promise(99, 'reserve Taz 1,2') from <site_name>

4. **Code Requirements**

- You must implement your project using either Java or Python unless you obtain permission to use another language in advance.
- You must use UDP sockets for network communication.
- Use the IP addresses and ports specified in the knownhosts.json file. Do not try to obtain the IP address or ports from the container directly.
- When creating sockets/sending UDP packets, you must use the IP address of the destination host. Since Submitty uses a special network configuration, addressing by hostname may not work.
- You may not use any libraries that provide “server” functionality, e.g., an HTTP server library.

5. **Source Code Management and Build Requirements**

The source code management and build requirements are the same as for Project 1.

6. **Deliverables and Grading**

(a) To submit your project source code, you must follow the Submission steps in Submitty. This will check out the project files from your Git repository and copy them to Submitty for grading. Submitty will execute several simple auto-grading tests to verify that your application builds and runs in the Submitty environment and that your application meets the UI specification. These tests will be worth 10% of the project grade.
(b) Each team must also submit a 2 page project report. This report must be single-spaced with 1 inch margins. Font size must be 12pt or smaller. The report can include diagrams. The report should provide the following information:

- Details of your project design and implementation. This includes descriptions of your data structures, stable storage implementation, use of sockets, and use of threads.
- A description of how you implemented the Learners and why you chose the particular method.
- A description of your recovery algorithm and a discussion of when/how it guarantees the Paxos safety and liveness properties.
- For groups of 3: A description of your leader election algorithm and a discussion of whether and how it guarantees safety and liveness.
- For CSCI 6510: A description of your checkpointing implementation.
- A description of how you tested your project (be sure to consider failures and message loss).
- A description of how the project work was divided among the group members.

Project reports must be submitted in Gradescope by the project deadline. A substandard report can result in up to a 10% point deduction on your project grade.

If your project does not run on Submitty, you will receive a score of 0.
No late submissions will be accepted.