Distributed Systems - Project 2

Ryan Lin and Xiaotian Duan

Introduction

We created a distributed calendar application that uses the Paxos algorithm to maintain a replicated log. This application runs on AWS and has a total of five nodes, each of which is hosted in a different region. We used the same exact four nodes as last time, and added a fifth node named Earl which is hosted on an EC2 instance in Singapore.

- **Node 1:** Alan (Oregon) http://ec2-52-88-92-72.us-west-2.compute.amazonaws.com
- **Node 2:** Bob (Ireland) http://ec2-52-18-229-252.eu-west-1.compute.amazonaws.com
- **Node 3:** Carl (Tokyo) http://ec2-52-69-215-52.ap-northeast-1.compute.amazonaws.com
- **Node 4:** David (Sao Paulo) http://ec2-54-207-62-34.sa-east-1.compute.amazonaws.com
- **Node 5:** Earl (Singapore) http://ec2-54-169-235-208.ap-southeast-1.compute.amazonaws.com

Project Design and Implementation

We used Python to develop this application and all of modules used from the first version, which include the socket, threading, parse, and Flask libraries. We reused a lot of the code from the former implementation, in particular the classes, web-based user interface, and backend logic. We also added in the quick-create and universal reload features to the user interface for ease of testing.

The application has six threads. Their responsibilities are as follows:

- Thread 1: Flask web server -- serves up the user interface, listens on TCP 80
- Thread 2: TCP server -- leader election algorithm, listens on TCP port 40061
- Thread 3: UDP server -- paxos algorithm, listens on UDP port 40070
- Thread 4: Leader state machine -- updates state of leader election
- Thread 5: Proposer state machine -- updates state of paxos algorithm
- Thread 6: Synod algorithm starter -- sends “prepare” for events in queue

Our application uses different ports for localhost and AWS. For localhost, we use TCP ports 40061 to 40065, and UDP ports 40066 to 40070. For AWS, 40061 and 40070.

Stable Storage

We store the log in stable storage as log_N.txt. We also store the highest proposal number and variables (maxPrepare, accNum, accVal) of each Synod algorithm instance in the file named pn_N.txt. In this case, N referes to the ID of the node, ranging from 1 to 5. Please see Setup.txt in the submission zip file for how to run the server locally.

Leader Election Algorithm

For leader election, we used the Bully Algorithm. This algorithm is used to elect the leader node, which serves as both the distinguished proposer (DP) and as the distinguished learner (DL). Each node records the ID of the leader, its current state, the timestamp it last initiated an election, and timestamp of last heartbeat message. When a node starts up, it will start a new election. Whenever a node starts an election, it will wait up to 5 seconds for OK responses. If it receives no responses after 5 seconds, then it assumes its position as the new leader. When a node receives the COORDINATOR message, it will update its leader ID to the ID of the sender. Nodes periodically check if the leader is alive by sending a heartbeat message every 3 seconds. If this message was not successfully delivered, then the leader is down.

Paxos Algorithm

Each node is both an acceptor and a proposer. Our implementation of the Synod algorithm is basically exactly the same as the one shown in class. To achieve consensus, each event in the log is a separate instance of the Synod algorithm. Each event consists of an operation type (insert or delete), appointment data, node ID that created the event, and an event index indicating its position in the log, starting from 0.

The leader serves as the distinguished proposer. Every time a node creates a new event, it forwards the serialized event data to the leader. The leader stores all forwarded events in a FIFO queue and processes them using the Synod algorithm one by one. However, it only processes new events after it has patched its log and ensured that it's up to date. The leader does all of the conflict checking. It can reliably check conflicts once it has an up to date log and therefore an up to date view of the latest calendar. If it receives a conflicting appointment, it will drop the event and not process it. The leader also responds with an acknowledgement upon receiving a non-conflicting event. If a node does not receive an acknowledgement, it assumes that its forwarded event was not received, and it will add this event into a queue. When the leader comes back online, it will re-forward these events to the leader.

First it generates a unique proposal number, then sends a prepare message to all acceptors (including itself). The acceptors respond with promises containing their last accepted values, and the leader uses this to choose a safe value to propose. It then issues an accept request with this safe value and the new proposal number. If it receives a majority of acknowledgements, then it sends a commit message with the serialized event data to all nodes. When a node receives a commit, it adds the event to its log and updates the calendar accordingly depending on the operation type. Therefore, if an event is in the log, we are completely sure that it has been chosen.

The highest proposal number is stored in stable storage. When a leader generates a new proposal number (by incrementing the highest by 1), it broadcasts this new number to all nodes, who immediately save it in stable storage. Whenever a node recovers or comes online, it asks all nodes for a copy of their highest proposal number. It takes the maximum of these responses, and saves this into stable storage as its highest number. This is how we ensure that all proposal numbers are unique.

When a node comes back online, it will patch its log, regardless of whether it's a leader or not. During this time, it's in the "patching" state. When it's done patching, it switches to the "ready" state. Leaders cannot initiate the Synod algorithm for new events until they are in the "ready" state. Forwarded events are kept in a queue until its log is up to date.Patching the log is done by setting an integer representing the event index (position in the log) to 0 and then sending a "probe" request to all nodes asking if they have an event with the specified index, if we do not have an event in our log at this index. If the recipient does, it responds with the event data, otherwise it ignores it. The sender will them add this event to its log. We repeat this process by incrementing the index value by 1 until we don't receive any response within 3 seconds, at which point we know that we're at the end of the log and therefore done patching. This "probe-response" protocol is more efficient than running the Synod algorithm for each event with the index we might be missing, as it avoids a lot of unnecessary messages. It's guaranteed to work correctly because if an event is in the log of any node, this means that it already has been commited/chosen, which means that an instance of the Synod algorithm has already been run for that position in the log to achieve consensus.