David Chen & Hannah Deen
December 3, 2017
Distributed Systems & Algorithms

Project 2 Description

Dependencies:
To be able to implement and maintain site stable storage, the Jackson Project and the standard JSON library for Java were used to serialize the logs and accepter data structures.

The Design:
For the solution to the Paxos Twitter problem a java code base was created. The repository includes the network, the Synod algorithm, and site stable storage. We assume that all sites contain proposers/accepters/leaners.

Beginning with the network implementation, the server and clientGUI pair are created together but function individually. The clientGUI is user facing and sends the propose or accept messages (if the process is the leader). For N number of servers and clientsGUI (the number of ip addresses and ports dictated/available), each clientGUI has its own 'home' server. When the clientGUI is told to Tweet, block, or unblock it first creates a propose message, detaches from its 'home' server by closing the server and in/out stream connections. The clientGUI then loops through the non-blocked servers of all other sites Tweeting, then reattaches to its home server to wait for more user input. When a message is sent to a site, it is sent to the server. The server then processes the message (in this case 'propose'), and then spawns a clientSender (if it has decided to promise) which then sends the promise message (or the proper response) to the requesting site. This server clientSender messaging continues through the commit stage. Since clients can only be attached to one server at a time, but servers can have multiple clients this assures that all events can be delivered without networking deadlock. This implementation also takes advantage that a server client pair does not have to be permeant, but can be reliable. To handle the servers and clients being independent the implementation used threads to create a multi-functioning Twitter base ie the server and client are run together, but are able to function independently.

Objects and Data Structures:

Name	Proposer/ Accepter	Type	Purpose
blockTable	Site	int[][]	In memory block tracking
tweets	Site	ArrayList<Tweet>	In memory tweets
id	Site	int	siteId
log	Accepter/ Learner	ArrayList<Event>	Consensus Log
index	Proposer	int	Most recent consensus index

Name	Proposer/ Accepter	Type	Purpose
majority	Proposer	int	half+1 processes, hardcoded
maxPrepare	Accepter	ArrayList<Integer>	Synod Algorithm Max prepare value
accNum	Accepter	ArrayList<Integer>	Synod Algorithm AccNum value
accVal	Accepter	ArrayList<Event>	Synod Algorithm AccVal
lastProposed	Proposer	ArrayList<Integer>	Array for tracking proposal numbers that have been sent
hold	Proposer	ArrayList<Event>	temporary holder for Event while propose/promise stages happen
promised	Proposer	ArrayList<ArrayList<accInfo>>	Tracks promised messages received (proposer side)
asked	Proposer	ArrayList<ArrayList<accInfo>>	Tracks ack messages received (proposer side)

Execution:
The algorithms implementation begins when a user calls tweet, view, block, unblock, or a site recovers from a crash failure. Each of these events are denoted as an 'event' object that is stored within the log; event objects can have three types (0,1,2) which specify tweet, block, and unblock events respectively. If an event is a tweet event it contains a Tweet object which the message being sent. If it's a block or unblock event, a field called blockee specified the target of the action.

For each event, a Message object is first created. This message includes the proposing site's ID, the index in which the event would be inserted into the log, the type of message (propose, promise, accept...etc), the proposal number (generated by the last proposed number + 1 * max number of sites to ensure uniqueness), and the event that would be added.

To begin the Synod algorithm, a site receives an event from the user. The site then starts the propose, the proposing site hold a temporary value for the event and index where the event would be placed in the log. The proposal number is then set for this round and placed within the Message object. The msg is sent to all sites (including itself). Each site then processes the message to check whether if n > maxPrepare. If it is, the site then responds to the original site with a promise message (the same message object, but with a different msg type value). When the original site receives promise messages from a majority of total sites it then checks to see if all accVals are not null (assuring that the site has not promised the spot to another). If all promise responses are null, then we can assume that no process has reached a majority, thus it is safe to propose its own value. The accepters then respond with ack messages if appropriate.

Once the site receives a majority of ack messages, it sends commit to all accepters/learners to update their logs and the tweets or block table respectively.

Stable Storage:
To make stable storage a reality, every time a message is received that would alter an accepter data structure, a JSON file is created saving the state of the accepter. If the file already exists, then the old one is overridden. If a site crashes this file is then loaded to reinitialize the site. On recovery, the log and all accepter data structures are reloaded, and the site objects are recreated.

Leader Election:
Leader election is simplified. Every time a client/site wants to create an event, the algorithm determines who was the site that reached consensus on the previous log entry (the leader). If the client making the event matches the previous leader, then it's allowed to send its value directly using the accept function and proposal number 0. The Synod algorithm continues thereafter as normal.

Maintaining the Solution to the Paxos Problem:
Safety:

- Only a value that has been proposed may be chosen (decided as consensus value).
- Only a single value is chosen.
- A process never learns that a value has been chosen unless it actually has been (no changing a decision).

Every proposed event by the client that is entered (tweets/blocks/unblocks) is inserted into the log using the Synod algorithm. If no proposer get a majority of promises, then the proposed value fails (similar situation with ack messages).

Livesness:
The algorithm completes as long as a majority of sites stay up for the described 'long enough' for the algorithms to complete. The networking scheme allows for multiple rounds for different log entries to be executed at the same time.

References:
https://stackoverflow.com/questions/43406275/getting-error-java-net-socketexception-connection-reset
https://www.javaworld.com/article/2077234/java-se/generic-client-server-classes.html
https://stackoverflow.com/questions/15541804/creating-the-serversocket-in-a-separate-thread
https://gist.github.com/dannvix/5385384
http://www.cs.rpi.edu/~pattes3/dsa/Paxos2.pdf