George Tsitsopoulos
Zachary Maurer

Project 2 Report

Assumptions

We assume that all usernames are unique. Also, we treat all duplicate blocks and unblocks as mistakes.

Design Details

The basic structure of the program is similar to Project 1. Users are represented by **Sites**, which have member variables IP, port, site id, and name. IP and port and integer and string that store the IP and port of the user in question. Site id and name are an int and a string taken from the input.txt. The Site ID is the line number in the file, while the name is a human-readable name for the user. Each Site has a List of these Site objects, one for each in the input file. As with Project 1, this class implements Serializable in order to be stored on disk.

We store relevant data for the Site in a class called **SiteVariables**. It stores a variety of member variables, and a functions to act on them. First, it has an integer called numProcesses, which represents the total number of processes in accordance to the input file. It also has a Site called mySite, which is the site the variables are in.

The bulk of the information for the algorithm is stored in three CopyOnWriteArrayLists. These are thread-safe, allowing us to operate on them in multiple threads without issue. The first is writeAheadLog, which is the final representation of Tweets, Blocks, and Unblocks that we get from running Paxos. These are all stored as **TwitterEvents**. This is mirrored by an array called paxosValues. This one stores the variables needed for the synod algorithm in a class called SynodValues. These two have an integer that represents their size, logSize. The last one is an array of Tweets, called timeline. This is what the user sees when they type “view”, and is changed for blocks, unblocks, and tweets.

SiteVariables has methods in order to operate on its member variables. Firstly we have the standard gets and sets for each member variable. We weren’t very restrictive with what could and couldn’t be retrieved this way, because we knew these wouldn’t be used outside this project.

expandLog is a void function that takes in a integer representing the size requested. It will expand both writeAheadLog and paxosValues to the requested size. It is used to ensure that, when proposing a log value, the log is big enough to accommodate it. This function also properly increments logSize. As with project 1, we have hasBlocked, which given a name tells us if the given site has already blocked the user.

recreateTimeline is called whenever the timeline is altered, whether by a new tweet, a block, or an unblock. It uses the blocks, unblocks, and tweets to construct a timeline that the user would see.

printWriteAheadLog is mostly for development purposes, but is a simple printing of each element.

Lastly, SiteVariables has a pair of clear functions. clearPromises takes in an index in the log, and calls clearPromises on the SynodValue in the paxosValues array at the chosen position. clearAcks does a similar thing, calling clearAcks at the appropriate index

SynodValues is used to run the synod algorithm on a single index in the log. It represents the values in one index. It stores a accVal as a TwitterEvent, and accNum as an integer. It also has integers for maxPrepare, myProposalNum, and an index for the leader. Lastly, it contains a CopyonWriteArrayList of Promises and one of Ack messages. These are all used in the synod algorithm.

Outside of the normal get and set methods, SynodValues has majorityPromises. It takes in a total number of sites, and returns a boolean if it has received a majority of promises or not. There is also a similar function called majorityAcks for Ack messages.

Lastly, clearPromises and clearAcks calls the already defined clear method of the two arrays.

TwitterEvent is a parent class for **Block**, **Unblock** and **Tweet**. These are the three types of events that can appear in our log. TwitterEvent itself has a String for the type of event, for easy checking, as well as a DateTime timestamp. In addition, it has standard get and sets for these.

Block and Unblock contain the two users involved in each operation. The “blocker” and “unblocker” are stored as Sites, while the “blockee” and “unblockee” are stored as the string of the corresponding user. These classes’ methods include basic sets, gets, and a toString for output.

Tweet stores both a String with the message, as well as a Site of the user that created it. It also has gets, sets, and a toString method

PaxosMessage is a parent class for all the message we will pass. It has a Site corresponding to the sender, as well as an Integer for the log index the message is related to.

Accept, **Commit**, **Prepare**, **Promise**, **RecoveryInfo**, and **RecoveryRequest** each only have one notable method. onReceive is called when the message is received by the ListenerServer. It handles what the algorithm does with each message.

Ack has a special method for sending commits. It takes in a SiteVariable for the current Site, as well as a list of Sites to send the commits to.

UtilityFunctions is almost identical to project 1. It has functions that don't really fit in the other classes, largely related to recovery using files on disk.

How It Works

Similarly to Project 1, our **Initializer** class takes in user input that represents which site is being created and starts both a **TwitterServer** and **ListeningServer**. The class also checks to see if the site is recovering from a failure or not. To do this, it checks the .dat file we store data in. If the file has data in it then the class will load the data into memory so that it can continue as normal. In order to assure there are no holes in the log, the class will then send a **RecoveryRequest** message to the next-highest-numbered site, or the first site if the last site is the one that failed. This will return the accNum stored at each log entry in a **RecoveryInfo** message. If the accNum returned for a given log entry is greater than the accNum at the failed site’s log entry, or if the entry doesn’t exist at the failed site, it will initiate full Synod at the log entry. This will make sure that the site’s log is up to date with the others.

The **TwitterServer** is a class that is run on a new thread. It listens for any user input, determines if the input is valid, and acts accordingly. If the user enters “view” then the class will print the timeline stored in memory. It will not contain any tweets from users who’ve blocked this site. If the user enters “view log” then the class will print all events in the write ahead log. If the user enters “tweet”, “block”, or “unblock”, the class will execute the Synod algorithm for the entry. If the site is the “leader”, which is determined by the site in the previous log entry, then it gets to skip the prepare-promise phase of the algorithm. Otherwise, it runs full Synod. To do this, it creates a **Prepare** message and creates a **SendMessageThread** thread. This thread will determine that the message passed to it is a **Prepare** message and then send **Prepare** messages to all sites. A timer is set for 5 seconds such that it constantly checks the number of **Promise** messages received so that if a majority of sites don’t return the message, a new proposal is created with a higher proposal number. The **Prepare** message is received by the **ListeningServer** thread on all sites. A new **OnReceiveThread** is spawned where we determine the message is a **Prepare**. This calls the “onPrepare” function in the **Prepare** class. This function determines if the proposal is large enough for the site to send back a **Promise** with its highest AccNum and AccVal.

Once a majority of sites return **Promise** messages, the createAccept function is called in **SendMessageThread**. This function will determine the correct value to send and create the **Accept** message. A new **SendMessageThread** is created, where the message passed in is the new **Accept** message. Similarly to before, the class determines the type of the message and sends it to all sites. It again waits up to 5 seconds to receiver **Ack** messages from a majority of sites. When a site receives an **Accept** message, **Accept**’s onReceive function is called where it determines if the n is large enough to respond. If it is large enough, it will send an **Ack** message back. If a majority of **Ack**s aren’t received within 5 seconds, the **Accept** phase is restarted. Once a majority is received, a **Commit** message is created and sent to all sites. When a site receives a **Commit** message, **Commit**’s onReceive function is called where the value is saved to the site’s write ahead log and the leader is updated. It also saves site variables to the hard drive using the **UtilityFunctions** class.