Project 1 Report

Ian Gross
Ben Mizera

<u>Basic Design Documentation</u>

- main.go
    - The program starts here and takes in the command line arguments (the entry data and the id of the site). The node to contain data for execution is created, a new thread (in function) to listen for connection requests and go to the function for accepting user input.
- node.go
    - The code is structured around the node struct, which contains all critical data, as well the mutex lock for that critical data.
    - The node struct contains the site id, the current counter value, the Log stored in memory (array of array of event structs), the Dictionary stored in memory (map[int] of map[int] of bool), the Time Array stored in memory (array of array of ints), the listen port number, the other site ips (map[int]string), and name of each site (map[int]string).
    - The functions in this file consist of node creation, load the Log/Dictionary/Time Array for the disk, write to the physical Log/Dictionary/Time Array on disk, has received event, increment local clock, clean the dictionary, and truncate the log.
- tweets.go
    - The tweet file contains the structs for the messages and the events that are stored in the log. There are various helper functions like getting values from the event or preparing the events to be sent or received to/from other sites.
    - The message struct contains the time array, the log and the send id.
    - A event is saved in the tweet struct. All events, even a block or unblock, are saved as a tweet struct. The tweet struct contains the message (if it's a tweet event, otherwise empty), the user id associated with the tweet, the follower (for block and unblock), the tweet time in utc, the counter at the site, and the event type (tweet, insert, delete).
- listen.go
    - The listen function is a separate function running as a thread. It accepts connections via the golang net package listen function using tcp. Once a connection is found and accepted a new thread is created to handle the connection in a separate function. This function creates a byte buffer to receive the message and send the data to the receive function.
- receive.go
    - This file handles all handling of the received messages. The mutex is called to prevent multiple writes to the log, time array or dictionary. From there, it will

        check which events are new using the has received function. These new events are stored in the log and dictionary and updates the time array accordingly. Once complete, truncation occurs on the log.

- Input.go
    - The file’s primary function is to handle user input and go to the appropriate function in the input file or a separate file. As specified in the homework, a user can tweet a message, view the tweets, block a follower or unblock a follower. Separate functions to print the dictionary, print the log and exit the program are implemented for the purpose of debugging.
- send.go
    - Contains two functions for sending out events to other nodes. *Broadcast* attempts to establish a connection with each other node, and if it’s successful, *Send* generates the message to be sent through the connection, containing the Time Array, and all new events, and sends them out. The message contains all messages that the sending node thinks the receiving node hasn’t received, contained in a large json.

Project Walkthrough

The program begins in *main.go*, where it accepts parameters for the entry data and the user id. The entry file is a json file that contains the names associated with each id, the number of nodes, and the ip:port number for each id. The main function then makes the local node, creates a separate thread to listen for incoming connections from other sites and enters a function to handle user input.

The node creation occurs in node.go. A node basically stores information needed for the various functions of the program. A node stores local id, the current counter, the username, listenport, ip targets, names of all users, a log (in memory), a timearray (in memory), a dictionary for block and unblock (in memory), and mutex. Some of these values are populated by reading the entry data. The other values are left empty (space is allocated) if a physical log or dictionary don’t exist already. If they do exist, they read from them to update the data in the node, which handles the case of crash recovery.

The listen thread handles the connection from other sites using golang’s net package (*listen.go*). When a connection is found, a new thread is created to handle this connection. It handles the read of the message and passes it to *recieve.go*’s receive function. Receive will lock the mutex, to prevent multiple actions writing/reading log, dictionary or time array. Then, it will determine which events are new using the hasreceive function and update the the dictionary, log and time array accordingly. The hasreceive function determines if a particular event’s counter is less than or equal to the time array for the specific user id.

The user input section occurs in *Input.go*. It recognizes commands for tweet, view, block, unblock, print log, print dictionary, and exit. A call to tweet a message increments the local clock for time array, creates a tweet event for tweet, appends the tweet to local log in memory, writes the log to physical memory and broadcasts it to the other ip targets. A tweet is defined the struct to contain information for a tweet event, insert event, or delete event. It contains fields to store the message (if a tweet event), the user id, the follower id (for insert and delete), the current time in utc of event creation, the event count number, and the event type (tweet, insert or delete). Write log occurs in *node.go*, where it saves the log in memory to the physical log json file. The broadcast function occurs in *send.go*. This function checks to see if the id is blocked (won't send to location) and uses the net package's dial function to connect using tcp to that ip. Next, it proceeds to send the log as a message, saves it as bytes and sends it to the connection site.

Block user command takes the id of the follower to block. It checks to see if the call is valid first. A non-valid command might be a call to block yourself, blocking a user that doesn't exist, calls block on a user that's already blocked or calls unblock on a user that hasn't been inserted. If valid, it will create a tweet event for insert, update the local clock counter, save it to the log in memory, insert the user follower pair into the dictionary, write to the physical log and write to the physical dictionary. The unblock command works in the same way as block. The differences include a delete from the dictionary, a different event type for tweet, different criteria for a valid unblock (mentioned previously).

The view function prints out all the tweets from every site in order of the most recent time associated with each event. The time is stored in utc, so it's standard across all sites. The print log and dictionary prints out each item respectively, for the purpose of demoing the content. The exit command exits the program gracefully by shutting down the threads before exit (not for the crash in the demo).

Wuu-Bernstein Implementation

Once the program is started, we are reliant on other nodes or the user for input. Whenever the user tweets, the program logs that tweet as an event, and increments the clock. Similarly, if the user blocks or unblock another user, we check that the action is valid, increment the clock, add the event to the log, and the entry to the dictionary. In the case of unblock, we eventually remove the entry from the dictionary in preparation for truncation.

After a user tweets, the system first tries to connect to each other node. If that is unsuccessful, we assume the other node is offline, and don't prepare a message. If we are successful in connecting, we prepare a message containing all events that we don't know the other node as received, as well as the local clock array, and send that, as a large json, to the other node. This message is then sent.

When the other node receives this message, it parses through the events, those which it has not seen before it adds to its own log, otherwise it discards them. It updates it's Time Array with the max values from itself and the sending node. It also adds or deletes anything from its dictionary if there's an event requiring it. After this, it writes the Log and Dict to disk to work against crashes.

This program follows Wuu-Bernstein as desired, with the more difficult bits of sending and receiving implemented with for loops.

Justification of Log Truncification

Truncation ensure that the system isn't carrying around unnecessary data. The spare bits and bytes of memory that would might be used in storing something that' already been done aren't wasted in that storage.

In order to truncate the log of insertions that have been deleted, the system uses a map of map of bools. This allows the system to store a user blocking multiple other users, giving instant lookup of the user's blocked, and clean deletion of the users from the dictionary when desired. In this implementation, when a unblock is called, the value is not removed from the map immediately. Instead, the bool value is set to false, indicating that the block in question is no longer active and can be truncated from the log. When broadcast is called, will only prevent sending to nodes whose value are true in the dictionary.

At the end of each recieve, the systems calls *CleanDict* which attempts to truncate the log. For each entry in the dictionary with a false value, we attempt to remove it from the log, given that every other node has seen both the insert and delete events.

Thread Implementation

For our solution, we utilize the main process thread and one additional thread. When the program executes, the main program goes to receive user input from the command line. Before this occurs, we run a goroutine on a function called 'listen' to handle the incoming connections concurrently with the main program runtime.The listen function utilizes golang's net package, which provides an interface for network I/O like TCP/IP, UDP, domain name resolution and Unix domain sockets. We used net.Listen, which creates a server to listen using tcp on the specified listen port number. Once a connection is received and accepted, a new golang thread is created to handle the connection and read the message, while the listener thread goes back to waiting for and accepting new connections. The receive function takes this message and handles storage in the local log and dictionary.

Prevention of Duplicate Inserts

To start off, we implement the log as an array of array of events. Each array represents the content stored at each site id. When a user calls a tweet command, the information is stored in this my site's data array and saves it in a permanent json file. A timeArray (an array of int arrays) is used to track the count time each site is at (like a vector array). For the dictionary, the information is stored in a map of maps of booleans. Just like the log, the dictionary is stored locally in memory and in a physical file.

To problem occurs of multiple inserts when receiving messages from one to multiple sites or the local site tries to send a message. Our solution was a mutex is for the log, dictionary, and time array to prevent more than one resource from writing to any of these objects at the same time. These mutexes are locked whenever a message is being sent or received. On the receive portion, it uses the hasreceive function to determine based on values in the stored time counter which values aren't in the local log yet. This prevents the same tweets from being stored multiple times. Additionally, with the dictionary being stored in a map, which can only contain unique identifiers. So if in my map for id 1 I have id 0 set to false, I cannot insert 0 again because the key needs to be unique.

The log and dictionary problem states that to maintain a dictionary or log over multiple sites, given the execution of E, if events e and f are are part of E and e happens before f, then the event record of e must be in the log of f. The event record of e must be in the log of site f and a delete request cannot occur without an insert request. Our solution solves this problem because the knowledge of one site is passed down to others on each send request and doesn’t assume the knowledge has been received. For the dictionary, any attempt to delete something from the map block checks to see if the element exists before delete, which prevents the issue of a delete before insert.