Distributed Systems and Algorithms

Project – 1  
Shruthi Chari  
Ananya Subburathinam

Our implementation of is designed to run from the command line interface. The user is given four choices, they can choose to send a message, view their timeline, block someone, and unblock someone. These are the events that occur in our system. For each tweet, we execute the synod algorithm for that particular slot number. Once the majority is gained, we commit the value in the log for that slot. If the same process tweets again, we just send accept (n, v) to all other acceptors and commit the value into the log. The implementation details of these functionalities are discussed in the following paragraphs.

Each **Event** in our project is represented as an object of the Event class. Its members contain the operation type (BLOCK, UNBLOCK, SEND, etc.), the counter Ci at the time of the operation, and the contents (such as tweet text, who was blocked, or who was unblocked), the node at which the event originated, and the physical timestamp (in UTC+0).

The Event class is built to be comparable and is hash structure safe. It also allows two separate methods for sorting: by the combination of the event’s node and the counter Ci, as well as by the counter Ci equalities are resolved by the associated timestamp. This class is the basis on which the log of our project is built.

We implement the synod algorithm initially to gain the majority from all processes. We start by sending a prepare message to all acceptors. We have used enums to specify the type of message being sent. Different categories of messages are PROPOSE, PROMISE, ACCEPT, ACK, and COMMIT. So we start by sending the PROPOSE message to all the acceptors following which the acceptors recognize the PROPOSE message and reply with the PROMISE. On receipt of majority of PROMISE messages, we validate the correct ACCEPT message with value (accNum, accVal) and send it to all the acceptors. Following this, the acceptors return with the ACK message and all the proposers commit the value into their Logs.

This class also implements the methods to store the log into stable storage in the form of a json file, and to load it back from stable storage into active program memory. The functionality of insertion of event, insertion of partial logs, deletions, searches are also implemented within this class to accommodate the Event objects. The *addToEventLog()* function allows for truncation with respect to the block and unblock operation. When a user initiates a block event, past blocks and unblocks targeting the same user are replaced by the newly formed block event. Similarly, when a user unblocks another user, past unblocks targeting the same user are replaced. In this case however, we keep the event log of past block operations. The sorting of the contained events is implemented here.

The dictionary is represented in the **Dictionary** class of our implementation. It holds the (user, follower) information of all the block events that are still active (no unblock has been performed for the same (user, follower)). This is done as a Map for the sake of program efficiency and quick lookups and insertions. The key of our Map corresponds to the user name of each blocked user, associated with a value which is a set of all users they are blocked by. The Map and Set structures naturally protect against duplications and resolve *insertion*, *deletion* and the *contains* queries fast. These functionalities are implemented within this class to accommodate the <user, follower> concept.

This class also implements the storing and loading the dictionary to and from a file stored on disk. The dictionary is kept as a JSON file, which is a lightweight data-interchange format, easy for humans to read and write and easy for machines to parse and generate.

The class **Message** functions as a structure that contains all the data to be sent across the network. Its fields include the user identification sending the message, the contents of the message as a string, the partial log calculated with regards to the destination target and the current node table. Objects of this class will be converted to the JSON format before being transmitted, and upon reaching the receiving client, it will be converted back into a binary object.

The class **NodeTable** is used to encapsulate the time table Ti which maintains information about process Pi’s knowledge of all other processes in the system. As mentioned above about other classes, we have boilerplate saveNodeTable and loadNodeTable functions to write to stable storage and read from it respectively. The getter and setter functions are used in the User class to update the table during the send, receive, block and unblock operations.

The class **User** is the main class in the program, which is used to implement all of WuuBernstein’s functions in tandem with the usage of the Log, Dictionary and NodeTable data structures defined. We have implemented the standard send, receive, block, view timeline and unblock operations. The thread implementation discussed below is also implemented in the same class.

**Thread Implementation:**

Since as a part of the Twitter simulator application requirements, we need every user to be able to send tweets out to every other user. We have adopted a multithreaded server as well as multithreaded client approach, let’s assume a connected network of N users; then this entails that every user is connected to (N-1) users. We can clearly see that this means that each user needs to serve as a server for all other N-1 clients, and each user needs to be able to talk to N-1 servers; so we need N-1 clients as well. Since we have the restriction of using a one point of execution, we decided to spawn user and client threads within our User class itself. Paraphrasing the above, a User class contains the following threads:

1. Server Listener thread that is waiting on incoming connections. This was used to avoid blocking the application when the main User thread is running its menu driven section.
2. N-1 Client Processing threads, these threads are spawned when there is an incoming client connection and they attach themselves to the main server thread pool.
3. Client Connection thread which is constantly trying to connect to N-1 servers. If a server is a down, a client thread retries every 1 millisecond for a reconnect.
4. N – 1 Client threads which act to listen to N – 1 Servers and anything they tweet.

Every time a server goes down, a client is notified on its client thread and it attempts a reconnect on the same. We have altered our basic understanding of the traditional client-server model, by designated our client threads to be listeners and server threads to be the senders. Messages are sent to blocked users but they aren’t shown in the in memory timeline. When there is a message to be sent, the main User process invokes the corresponding client worker thread and sends out content on the same.

**The Log Problem:**

The log problem states that for every event *e* and every event *f* that happens before event *e*, *f* →*e* *iff f*R ϵ L(e). This is handled in our implementation as follows. For every event that happens, we add a record of it into our log. On a send of a message, we construct a partial log of all event records that each connected user must receive based on what they know (from our node table Ti). When each process receives a message with a partial log, it will incorporate all new records into its own log (and perform truncations as needed). This ensures that eventually, even considering message loss scenarios, the log problem will be solved.

**The Dictionary Problem:**

The dictionary problem is defined in the Wuu-Bernstein paper as: x ϵ V(*e*) *iff* cx → *e* and there does not exist an x-delete event, *g*, such that *g* → *e*. Here cx denotes the insert(x) event. We handle this in our implementation of the Dictionary class. Insert events happen by querying the Dictionary class object which is a Map data structure. The delete event is reflected in our code as a “remove” function that removes a <user, blocked> association by querying the Map which stores the dictionary with the “user” as the key, then it accesses the set of all this user’s blocked users and removes “x” from the set of blocked users. The insert function inserts the new blocked user into this set. Since a set does not allow duplications, each blocked user associated with a user will be unique.

**Language Used:**

This project is coded using java 1.8.

**Classes Used:**

1. Dictionary
2. Event
3. EventLog
4. Message
5. NodeTable
6. OperationTypes
7. SetExample
8. TrialClient
9. TrialServer
10. Tweet
11. User
12. UserClientConnector
13. UserConnector