George Tsitsopoulos
Zachary Maurer

Project 1 Report

Assumptions

We assume that all usernames are unique. Also, we treat all duplicate blocks and unblocks as mistakes. This allows us to essentially ignore them, as we keep the state of the system (log,dictionary,clocks) unchanged when one of issues occur.

Design Details

Each user is represented by a **Site** object. The class Site implements Serializable and has four properties: an IP address stored in “ip”, a port number stored in “port”, a site id that corresponds to its line in the configuration file “id”, and a human name to represent the user’s “name”. This is all of the basic, relevant information that defines a site.

The main variables stored for each site are stored in an object of type **SiteVariables**. SiteVariables implements Serializable and has a number of properties and methods. It stores the dictionary in a variable called “dictionary” of type ConcurrentHashMap<Block, String>. Each key describes a unique block event. Each value is arbitrarily set to “True”. We chose to use this structure because it is safe when multiple threads are involved so we wouldn’t have to worry about concurrency issues with it. We store the partial log in a variable called “partialLog” of type ConcurrentHashMap<LogEvent, String>. The value is always “True” just like the above case. We used a ConcurrentHashMap for the same reason. A third property of SiteVariables is the int “localClock”. This is a simple counter that is incremented upon the input of a valid TwitterEvent. We store the matrix clock in an AtomicIntegerArray called “matrixClock”. We chose to use this type because of its atomic nature (thread safe). We have functions that allow us to get or update a value in “matrixClock” at the bottom of the SiteVariables class. The last two properties are the total number of sites in the system (“numProcesses”) and “mySite” which stores the site’s Site variable. The class contains functions to add to “partialLog”, add to and remove from “dictionary”, and tick both clocks. It also has functions to see if a user is blocked. It has a getNP method that creates NP by calling a hasRec method that compares the “matrixClock” values. The final method is a printMatrixClock method.

We created multiple objects for all the different types of events that can occur in this system. We defined three types of **TwitterEvents** - Tweet, Block, and Unblock. Each of these events extend TwitterEvent which has two properties (String “eventType”, joda.time.DateTime “time”). A Tweet has a Site “user” and a String “message”. A Block has a Site “blocker” and a String “blockee”. An Unblock has a "Site “unblocker” and String “unblockee”. These structures are able to hold all the information regarding the three main functions of Twitter.

In order to store the events in the log, we created a class called **LogEvent**. It has an int “id” representing the site id where it was created. It stores the local clock time at which it was created in the int variable “localTime”. Finally, it stores the TwitterEvent that was created in the variable “event”.

A **TwitterMessage** is an object that contains a Tweet “tweet”, AtomicIntegerArray “matrixClock”, and a ConcurrentHashMap<LogEvent, String> “np”. It is modeled after the content of a send message in Wuu-Bernstein.

The **UtilityFunctions** class is exactly what it sounds like, as it has some methods that don’t particularly fit in any of our other classes.

How the System Works

The **Initializer** class accepts an integer argument that serves as the Site id. It parses the configuration file to create all other Sites and then spawns two other servers - a TwitterServer and ListeningServer - passing our list of sites to each.

The user interfaces with a class called **TwitterServer**.It is an infinite loop that prompts the user for input. The defined commands are tweet, view, viewlog, viewdict, block, and unblock.

The **view**, **viewlog**, and **viewdict** commands all display data from the partial log or dictionary. The view command will only display tweets by users who we believe are not blocking us. It is the “client-facing” view command. viewlog views the entire log. This includes all tweets, as well as all blocks and unblocks that we have not yet removed from the log. viewdict displays the dictionary of blocks.

The **block** and **unblock** commands will either insert or remove a block event. It automatically checks if the target user is valid (is in our list of sites, isn’t a duplicate command).

The **tweet** command takes in an argument that is a string, which is the message of the tweet. These are stored in a Tweet object. Upon reception of a tweet, NP is calculated. The tweet, NP, and the matrix clocked are wrapped in a TwitterMessage and sent to TwitterClients.

A **TwitterClient** is spawned for each non-blocked site. This allows us to parallelize the sending of the tweets. They are designed so that, should the socket fail to connect, the program will not crash. The a serialized version of the message is sent. The tweet will be received by the follower at some point in the future in someone’s NP that is sent over.

Before the TwitterServer loop returns to its waiting state, we write our SiteVariables to an external file to be recovered in case of a crash. We serialize the object to store it.

Listening Server is the other server spawned. It has a “serversocket” to listen on and a copy of the SiteVariables. It has an infinite loop that waits for twitter messages, and, on reception, processes it.

NP is added to the current log (duplicates will just be replaced). The dictionary is then updated in a way that makes sure that the block/unblock events are applied in order. This assures that a user can’t get blocked twice if we receive an unblock and block of an already blocked user.

The matrix clock is then crossed with the one given by the message in accordance with the Wu-Bernstein Algorithm. This ensures that the Site has accurate information on who knows what events.

The partialLog of the current site is then truncated. This is done by checking to see if, for any given block or unblock event, if this Site knows that all other Sites know about it. If so, the event can be safely deleted from the log.

Lastly, the site variables are serialized and stored in our file again.