# Tweeting with Paxos

Kathryn Lovell, Eugene Umlor, Rensselaer Polytechnic Institute

## I. User Interface

The user has access to several commands in this program.

- tweet $< content >$: the user will create a tweet event, and initiate a Paxos algorithm until it can be added.
- block $< user >$: the user blocks another user, creating a block event which will be replicated at all sites, and meaning that any subsequent view commands will not show tweets by the blocked user. The user can block any and all other users, including itself.
- unblock $< user >$: reverse the block against a user.
- view: print out all tweets from non-blocked users.
- exit: crash the server.
- reset: close the server and clear the log.

## II. Design Patterns

A single thread handles the user input, AcceptKeyboardInput. Similarly, a single PaxosServer thread handles accepting connections, and spawns threads for each connection. There is a thread for each InputStream opened by a connection, called RecvMessages. When the instance receives a message, it adds that message to a queue of messages _qMessages on the Paxos instance, which are then handled one at a time in the HandleMessages thread, which can handle PREPARE, PROMISE, ACCEPT, LEARN, and COMMIT messages. The log is stored as an array of EventRecord objects, consisting of an operation (NONE, TWEET, BLOCK, UNBLOCK, DUMMY), a username, some content, the wall clock time, and the user ID.

In our code, the functions of the proposer and acceptor are both performed the same Paxos instance, they are not separate. Simply, the Paxos instance is neither a proposer nor acceptor alone, and the terms proposers and acceptor are used for clarity in similarity to the Paxos algorithm paper.

## III. Log Entries and Full Synod

The index of this EventRecord's position in the log is used to ensure Synod operates solely on that logIndex. Each message is tagged with the logIndex it is operating on, and therefore which instance of Synod it is applicable to, as Synod consensus on different logIndexes can operate concurrently.

## IV. The Paxos Algorithm

The full Paxos algorithm is triggered by the user creating a TWEET, BLOCK, or UNBLOCK event. The thread chooses a proposal number using the nextHighestPropNum(int) function, which chooses the next highest multiple of n (where n is the total number of servers), and adds the site's ID (a number 1-n) to guarantee that two sites will not generate the same proposal number if passed the same integer. There is an exception to this, in the case where a site is the distinguished proposer, in which case it will simply send with a proposal number of 0. First, we send a PREPARE request of the proposal number we have chosen. When each other site receives a PREPARE request, it will respond with a NONE value if it has not accepted anything, or a PROMISE including the value if it has. When a site receives $> n/2$ promise messages (where n is the total number of sites), it sends an accept message to all sites with either the event it itself proposed, or the value in the promise with the highest proposal number. When an acceptor receives an accept command, providing the proposal number of that accept command is greater or equal to the one it has prepared for, it will lock in that value and accept it, which will never be changed (though the accepted proposal number can increase). After accepting, the acceptor will send a learn message to all proposers with the value it has accepted. When a proposer receives $> n/2$ learn messages with the same value, it will instruct all acceptors to commit that value to the log with a commit message. If the event is not a DUMMY, then the event will be added to the log at all sites. DUMMY events are used to

aid in recovery, so we will discuss them later. After COMMITting the value to the log, the site will reset all non-log state variables to their initial values in the Synod algorithm, so that Synod on a different entry can be dealt with. Multiple Synod instances can be running on the network, but only one Synod instance can be running on a site at any time.

When a user creates an event, that event is added to a queue of events. It is not removed from that queue until the site has successfully COMMITted that event to its log (or, in the case of a DUMMY event, caught up). This means that events can only be lost if a site crashes before the COMMIT process finishes. Otherwise, the server will continue to attempt to send that message until it has COMMITted.

## V. Event Types

Our program supports three functions - tweeting, blocking, and unblocking. Each are treated as Events with respect to the log, which means that all three will be recovered on crash. When a user blocks another user (or itself), that user will no longer be able to see tweets from the blocked user, though those tweets will still be included in the replicated log at that user's site. This can be undone with the block command.

## VI. Sites

We have five Amazon EC2 instances set up, with four in Virginia and a fifth in London. All instances work exactly as expected.

## VII. Recovery

The log entries, received promise messages, received learn messages, and state variables maxPrepare, accNumber, and accValue, are stored in stable storage, each in separate files. The Paxos instances attempt to recover on startup by first parsing these files, to see if there is anything stored from previous executions, and then initiating a dummy message. The other instances will respond to this DUMMY proposal by executing full Paxos. If the DUMMY message makes it all the way to being COMMITted, then we know the recovering site has caught up to a reasonable degree, and the DUMMY message will be ignored when it would be added to the log. Without attention, sending and waiting for input from a crashed instance can significantly slow down the rest of the processes, when it doesn't crash them entirely. Thus when an instance crashes, the other instances must somehow detect it and avoid sending messages to it. To do this, we place a timeout on the OutputStream that is attached to each instance. If this OutputStream.readLine() returns null for a set time, we know that this instance is non-responsive, and we close and remove the socket from our list. When the crashed instance recovers, then, it can safely re-create its connection to each instance.

## VIII. Known Issues

Occasionally, when a site attempts to create an event without $> n/2$ other sites up, the Paxos algorithm will hang, and no events can be committed even after $> n/2$ sites eventually do come up. We are not certain of the cause, but it likely has to do with the server not sending PREPARE messages to sites that come up after it has sent the first round of PREPAREs, which would be a somewhat simple fix.