Distributed Social Networking Application with Paxos

Andrew Showers, Dylan Elliott, Yi Liu

December 3, 2017

This project demonstrates the implementation of a distributed social networking service by maintaining a replicated log using the Paxos algorithm which uses the Synod algorithm for consensus on the contents and ordering of the log. The log at each site displays the activity of all users such that no log disagrees in the causal order of events.

1 Implementation

We program this project in Python 3 and use the Python `socket` library for socket programming with UDP protocol and the Python `_thread` library for multi-threading. The Paxos algorithm is implemented such that each machine has one instance of a *Proposer*, *Acceptor* and *Learner* that are configured to communicate on different socket ports in order to behave "virtually" as separate machines. On start-up, each machines Proposer, Acceptor and Learner receives their corresponding port assignments from a `hosts.txt` text file containing this port information as well as IP information and machine user-name information. Therefore, prior knowledge is provided such that each machine is aware of all others and knows how many machines are required to form a majority for the consensus problem. We deploy the Proposer, Acceptor and Learner "nodes" of the Paxos algorithm as a separate threads launched from a parent process that manages a terminal UI. A user interacts with the application using the commands *tweet*, *block*, *unblock* and *view*, to post test to the log, block a user from seeing their posted text, unblock a user from seeing their posted text and to view all the posted texts in the "timeline", respectively. Additionally, we include the commands *blocklist*, *log*, *servers*, *drop* and *exit* for displaying the block-list, displaying the full log, displaying server information, forcing dropped messages (for debugging) and gracefully exiting the program. Below we describe the functionality and implementation of the main components of the application.

Events: Events are implemented as a Python class containing meta-data for the event and the raw data the event holds (if it is a *tweet* event). The three events passed between processes in this system are *tweet*, *block* and *unblock* events. The *tweet* event contains meta-data such as the UTC time the tweet was posted as well as the actual string data that is the subject of the tweet. The *block* and *unblock* events consist of a (username, follower) pair, indicating that "username" is blocking "follower" from viewing their tweets.

Log: The log is implemented as a Python class and contains a list of the event objects described above. Every time an event is added to the log, we append the new event object to the list and also append this object to a log backup file on disk. We use the Python `pickle` library for easily handling our event and log objects as well as for the ability to write-append to files on disk for back-up. The log also contains a *timeline* and a *blocklist* showing the tweets that are viewable to the current user and the information about who-blocks-who in the system. Every time the log detects that a block or unblock event has been added, it calls a `rebuild_timeline` method that constructs the timeline of viewable tweets according to the who-blocks-who list.

Proposer: The proposer is implemented as a Python class, and on creation by the main program it starts three threads. The first thread opens a listening socket and a received message buffer and listens for any incoming messages, the second thread initiates a garbage collector to clean messages that have been handled from the message buffer and the third thread is another maintenance thread that attempts to fill any missing information (holes) found in the log (we explain the implementation of this thread in the "Holes in the Log" section). On retrieval of a message, the listening thread then spawns a new thread to process the message and add it to the message buffer. The proposer class contains an `insert_event` method for attempting to add either a *tweet*, *block* or *unblock* event to the log. When the user would like to perform one of these events, the proposer class searches the log from the end for the next available open slot that is not a"hole" and then attempts to propose a value for this slot. The proposer then runs through the mechanics of the Synod algorithm, returning false if at any point there are failures. These failures mainly occur if the proposal is ignored by the algorithm or if messages are lost and therefore, because we are using UDP protocol,

a timeout is detected to confirm that failure. Upon failure, the proposer simply waits a finite amount of time and tries again for the same log slot. Eventually the proposer either learns that the slot is available and sends accept messages for its value or learns of another value that "won" for that slot and therefore further confirms that correct value through accept messages with an updated value. In the latter case, the proposer will still learn that it has failed to add its information to the log and therefore will propose its value again for the next available log slot—essentially starting a brand new instance of the Synod algorithm. This repeats until the proposer succeeds at finding an available log slot and when it receives enough acknowledgments from the acceptors after proposing its own value, it sends a commit message to the learner nodes in order to add that value to the log.

Acceptor: The acceptor is implemented as a Python class, and on creation by the main program it starts a listening thread. The acceptor is a passive component in the algorithm that only acts when it has received a message from the proposers, upon which it starts another thread to process the message and add to a message buffer. The acceptor follows the acceptor mechanics of the Synod algorithm for each log-slot proposal by responding (when the proposal number is large enough) to any proposals with "promise" messages and any acceptances with "ack" messages. The acceptors additionally always respond to a special proposal number (0, 0), that indicates when a proposer is attempting to learn a value and therefore should not be ignored according to the standard Synod algorithm. The response to these messages is identical to all others and therefore the acceptor will pass on any information it has about a value that has been chosen for a specific log slot.

Learner: The learner is implemented as a Python class, and on creation by the main program it starts a listening thread. Once again, this listening thread spawns a thread to process any messages it receives. The learner performs one simple task; if it receives a 'commit" message from any proposers, add the contents of that commit message to the slot in the log indicated by that commit message. At this point in the Synod algorithm, a proposer can only send a commit if it is either reinforcing the same information that may already be in the desired log slot, or if it is certain that the log slot is available for new information it is providing.

Data Saved to Disk: Every time the log is updated, the application is able to append the new contents to a growing copy of the log on disk. This is easily done using the Python `pickle` library since all of our events are implemented as Python classes. Additionally, each acceptor saves the data it uses for the Synod algorithm at each slot in the log because if a process fails during the proposal of a value, we would like to recover any saved meta data that prevents the process from letting a later value win that log slot. In order to avoid completely re-building the timeline and blocklist data structures from the log after a possible failure, we employ checkpointing of this data every time 5 new event entries are added. While the log is loading events from disk, the entries are loaded in chronological order due to appending to the file on writing. So the events loaded are added to a replay list, and the replay list resets after every 5 events. Once the file is fully loaded, the remaining events are those which would not have been saved in the checkpoint process and are "replayed" by reprocessing them to the timeline and blocklist. This way, we only have to "replay" a maximum of 4 events from the log to rebuild the timeline and blocklist structures after a failure.

Catch-up after Recovery: If a process is down for an extended period of time, the main thread is also configured to call the `update_log` method of the proposer on start-up after the the proposer, acceptor and learner classes are initialized. This method uses the (0, 0) proposer value mentioned above in order to learn missed log slots until it receives responses back that indicate that the next available log slot is empty. At this point the method terminates as it has become aware that the log is now up-to-date and we can continue to the normal UI.

Holes in the Log: Occasionally, because we are using UDP socket protocol, it is possible for holes in the log to form. Occurrence of these holes are checked every 60 seconds by a `hole_filler` thread that continuously runs in the background on the proposer class. This method calls another method which returns a list of all the slots in the log that currently contain holes. The proposer uses the (0, 0) value while iterating through the slots where the holes are located to learn the missing values using the standard Synod algorithm.

Optimizations: We reduce message complexity slightly by allowing the process who got their information into slot i of the log skip the proposal phase of the Synod algorithm for getting information into slot $i+1$ of the log—we call this process the "leader" for log slot $i+1$. Since each event entry of the log contains the ID of the process that committed that entry, it is easy to use the log to determine who is the "leader" for a specific log slot. The leader proposer is allowed to skip straight to the accept phase, allowing the same process to perform multiple events quickly in a row with low message complexity if no other processes squeeze in an event between that time. If the later occurs, the leader will simply learn of the event that got in-between and the safety of the algorithm is still preserved.