Implementation of Wuu Bernstein Log Algorithms in Python Using SQLite for Data Persistence

Lucien Christie-Dervaux, Matthew Obetz

October 15, 2017

This project is a distributed Twitter client that can be run on N machines. Each client can tweet, as well as block and unblock other clients from seeing those tweets. These events are stored locally in a log, and efficiently sent to other clients, who then add those new events onto their own logs. The algorithm used is an adaptation of the Wuu Bernstein Algorithm for replicated logs and dictionaries. The program is written in Python, using sockets to transfer messages and SQLite to maintain the log of events and blocks on disk.

Introduction

The Wuu Bernstein algorithms are a pair of distributed algorithms which operate on N processes in a connected graph, either maintaining a Log of events or managing insertions and deletions into a set (referred to as a Dictionary). Messages can be lost and clients can crash then later recover, but all algorithmic actions are atomic and there are no disk failures. To accomplish efficient transmission of messages, Wuu Bernstein also keeps a matrix of knowledge a process has of other processes' previously received events, making it so redundant messages are not sent as frequently. At the application level, we define the events Tweet, Block, and Unblock, where Block and Unblock correspond to Dictionary $insert(x)$ and $delete(x)$ with x being a $(blocker, blocked)$ tuple.

Program Architecture

The program is divided into three parts: a main thread to handle input, a logger, and a communicator, which are described below:

Executing main.py starts the main thread, reads the list of site names and addresses from a config file, then initializes a Communicator object and a Log object. The program then listens on the command line for a tweet, block, unblock, or view: Tweet informs Log to record the received tweet to disk, then instructs Communicator to broadcast the message once recorded. Block informs Log to add a received user to the Log and dictionary. Unblock similarly calls to Log to add an unblock event and remove the entry from the

dictionary. View asks the Log class for all the current Tweets in the database that are not from a user that has blocked this client.

The Communicator class in udp_communicator.py is in charge of managing the array of site addresses and ports. When initialized, it creates a separate thread that binds the port specified for this site in the config file, then listens for incoming messages from other clients. It also spawns temporary threads to accept tweets from the command line through main.py and transmit them to other clients. Both types of threads communicate with the static Log class to add to the Log and add or remove from the blocks dictionary. The messages being sent and received are Message objects serialized into JSON.

The Log class is a static class that handles atomic transformations to the Log and Dictionary as function calls. Atomicity is supported by creating transactions that modify a locally stored sqlite3 .db file. The functions support adding local tweets to the log, adding new events from received messages, and adding and removing local blocks from the dictionary. The receive function makes sure no duplicate events are added to the current Log, and that the timestamp matrix T is updated to reflect new knowledge of the knowledge of the other clients. Finally, at the end of all receive events, the Log gets trimmed of all block or unblock events in the log which all clients know about according to the current T matrix. Whenever Log needs to pass events to other classes, the rows returned from queries are converted to Event objects.

There are also Event and Message classes which serve as data structures for packaging event records and whole messages, with utility functions for serializing/deserializing and formatting output for users.

Satisfaction of the Log Problem

The Log problem seeks to find an algorithm where, for a collection of connected sites with local total ordering of events, given a partial casual ordering of events between sites, $e \rightarrow f$ implies $eR \in Log(f)$. That is to say, if event e *happens-before* event f, then any log which contains f must also contain e. For our application, recording a tweet in the Log is the total effect of that event occurring, so the Log problem is trivially satisfied for two tweets occurring on the same process – nothing in our program can delete from the Log where Log.op="tweet", so if $\neg e \in Log(f)$, then e does not and could never have existed.

The only other way to create a relationship $e \rightarrow f$ is on the completion of a message from $p_i \rightarrow p_j$, where $eR.site = p_i$ and $fR.site = p_j$. In our application, this is represented by Communicator.broadcast_message() occurring on p_i, followed by Communicator.message_listener() receiving that message on p_j and successfully completing Log.receive(). We ensure that by the time p_j executes f it has already recorded $\forall e(e \rightarrow f)$ using a regular implementation of Wuu Bernstein – each site stores, in addition to a Log of events, a two-dimensional timestamp matrix T as a table of $(site, knows\_about, timestamp)$ rows where $T[site][knows\_about]$ is the value $timestamp$, storing the number of events our process knows $site$ has previously received from $knows\_about$. On a send to p_j we transmit all events e where $eR.time > T[j][eR.site]$. (in other words, all events that have a timestamp greater than their site's value in the last timestamp information we received for p_j.) In Log.receive(), the receiving process stores all these events, and also takes a pairwise max of each element in the received timestamp with the current

value in its own timestamp (updating its knowledge of the outside world), as well as a pairwise max of the row in its matrix representing its own knowledge of itself with the knowledge the sending process had of itself (since it now cannot know less than the site who just sent to it.)

Since a sender transmits all events that it has not confirmed the receiver has already processed, it is impossible for a receiver to ever get a message that doesn't also include all the casually preceding events it has not yet encountered. This can be observed by the fact that site p_i is the only process which can create a new maximum for any value in the row $T[i]$ on the matrix of any process (since self-incrementing $T[i][i]$ on a local operation and taking the pairwise max of the values in $T[i]$ and sender j's $T[j]$ are the only two ways that one of these values can increase.) This means that until p_i sends out a new message, it will continue receiving all messages every other process has collected since i's last message. Since all of these received events are recorded, they will always exist in the Log by the time event f occurs.

Adaptation of Dictionary Problem

Surprisingly, our application does not satisfy the dictionary problem as formulated by Wuu and Bernstein. The dictionary problem seeks to find an algorithm where for dictionary V, $x \in V(f) \iff insert(x) \rightarrow f \wedge \neg delete(x) \rightarrow f$. That is, the dictionary after event f contains element x if and only if an insert of x *happened-before* f but a delete of x did not *happen-before*. However, the Wuu Bernstein satisfaction of this property relies on x only ever being added once, where in our problem domain x represents a block tuple, which users may wish to add or remove repeatedly.

By removing the uniqueness constraint, the property can be violated with only three events executed in sequence on a single process: $insert(x), delete(x), insert(x)$. At the end of this second insertion, $insert(x) \in PL \wedge delete(x) \in PL$, so according to the existing definition of dictionaries, at future event f, $\neg x \in V(f)$. However, in the context of adding a block, the ordering of events suggests that the user wishes for the block to be present in the dictionary of blocks. As such, the definition of dictionary must be modified.

We opted for the weaker expression of the elements in a dictionary: $x \in V(f) \iff \exists i(i \rightarrow f \wedge i = insert(x) \wedge \nexists d(i \rightarrow d \rightarrow f \wedge d = delete(x)))$. That is, x should exist in the dictionary V after event f if and only if there exists an insert of x that happens before f, but there does not exist a delete of x that happens after the insert but before f. This modification to the dictionary property also required a complementary modification to receive(). In classic Wuu Bernstein, during a receive $V = \{v | v \in V \vee insert(v) \in NE) \wedge \neg \exists d\,(d \in NE \wedge d = delete(v))\}$, where NE are the new events received in the message. However, given our update to the dictionary property, we must instead filter d to only negate those inserts whose deletes have a greater timestamp. In our application, this was implemented with the event helper methods superceding_unblock_exists() and superceding_block_exists() on the Event class. Given a list of events, these functions both return True if there is a later event which would negate the effect of the current event. We filter the events received to only those for which no superceding block or unblock exists, meaning that we will end up with only the most recent block or unblock for a given site combination if one exists in NE.