Distributed Systems

Project 1 Report

Karan Sarkar and Caleb Longtine

Our implementation of the distributed event planner uses two threads. The main thread handles terminal input from the user. The second child thread receives TCP messages sent from other machines. We felt that two threads were necessary because our program needed to be waiting for input both from the user and from other machines. If we alternated waiting between both input feeds, we could have ended up in a situation where our program was waiting for the next line of input from the terminal but many TCP messages were unread and vice versa. We also wanted to keep a semblance of the causal order of events, so that messages could be received at the same time as a command was inputted. Using multiple threads allows our program to respond to both input feeds as quickly as possible.

The biggest complication from a multithreaded approach is dealing with shared memory. Both threads need to access common resources such as the partial log, matrix clock, calendar, and list of hosts. We addressed this problem by using a lock. In our program, whenever a thread needed access to the shared resources, it would acquire the lock and subsequently release the lock upon completion. Note that only higher-level functions like schedule and cancel deal with the lock. Many lower-level helper functions like find\_conflicts and tcp\_send do not. They come with the precondition that the function calling them has already acquired the lock. Using these functions otherwise will cause undetermined behavior.

To handle TCP messages, we encapsulate the process in the tcp\_send and tcp\_receive functions. Both functions use Python’s pickle library to serialize objects. Serialization and subsequent deserialization allows us to send whole objects without any problems. The pickling is handled completely within the tcp\_receive and tcp\_end functions for encapsulation functions. The TCP helper function ensure that all the other function only have to worry about supplying the correct data rather than the overhead of handling a connection. tcp\_send creates a new socket each time it sends a message. On the other hand, tcp\_receive maintains a single socket. This discrepancy occurs because, read\_tcp, the client function of tcp\_receive, waits for tcp messages to arrive. Therefore, we need to have the socket available at all times.

Our implementation of the hasRec function from the Wuu-Bernstein algorithm is included in the prune\_partial\_log function. The prune\_partial\_log function, as the name suggests, encapsulates the task of pruning a log for known events. We noted that in the Wuu-Bernstein algorithm, we need to prune the log based on what a single machine knows and based on what all the machines know. In order to reuse code, we designed the function prune\_partial\_log to take the argument hosts which lists the machines whose knowledge of the untruncated log events we should consider. We then return all events in the given log that hosts does not know about. This allows us to call the prune\_partial\_log function several times under different circumstances. For example, when we broadcast an event to a process, we only want to send events that the process does not know about, so we call prune\_partial\_log with the process’s host name. We also want to remove events from the log that all other events know about, so we can call prune\_partial\_log with every event’s hostname, which removes all events all hosts know about.

The add\_event and delete\_event function are responsible for adding entries to the partial log and modifying the calendar. One key design choice in these functions was the decision to include the change\_host\_flag as an argument. This argument is a boolean which tells the function whether the host of the event should be changed to the current machine. In other words, the change\_host\_flag represents whether we are the creator of the log entry. If the current machine receive the log entry or event from another machine, for example, then the machine that sent the message should remain being the host. However, if the current machine discovered a conflict and sends out a delete corresponding to an event received from another machine, the current machine must change itself to the host and thus the owner of the new delete. Our event class is also used for logs, so we deep-copy event and log entries and change their attributes in order to be more efficient.

The change\_host flag also controls whether the log entry should be added to the current machine’s partial log. When we receive an add or delete from another process’s partial log, we extend our partial log to include new events. We do not need to add duplicate events unless the current machine has created a genuinely new event through finding a conflict. Not including this modification allows duplicate creates and deletes to pop up because a machine will receive a delete and then create its own copy of the delete later.

The change\_host\_flag also has a significant effect on the delete\_event function. When the current machine receives a delete message from another machine, it calls the delete\_event function with the change\_host\_flag as false. Normally, this means that the event will be deleted without the host or partial log being changed. However, it is possible that the current event has already deleted the event. Upon the failure of a delete, nothing will happen because the change\_host\_flag has indicated that the partial log will not be modified. This allows the program to handle potentially extraneous delete messages naively.

The conflict resolution system is nontrivial. One such case occurs when a newly created event conflicts with multiple different existing events. Our approach is to check the lexicographically first event from the set of events that conflict with the new event. If the new event’s name comes before the lexicographically first of the events it conflicts with, we will keep only the new event. If the lexicographically first event’s name comes before the new event, the new event will be deleted and all of the existing events will be kept. This process ensures that even if a new event conflicts with many events, there will be a unique resolution.

The longest function in our function by far is the read\_tcp function. Its purpose is to wait indefinitely for TCP messages to come from other machines and handle them. Most of the read\_tcp function is just following the Wuu-Bernstein algorithm directly and delegating to the helper functions described above. The one portion that differs from the algorithm involves the creation of the add\_list. The add\_list is an array local to the read\_tcp function that is supposed to contain the events that must be added to the current machine’s calendar. The add\_list is created by several loops through the received partial log. In the first loop, we add any events that have a create. In the second loop, we remove any events that have a delete. The resulting list is all the events that we must add to our calendar.

In order for a process to stay persistent, we pickle the matrix clock, calendar, and partial log and save it to a local file unique to that process. Whenever those values are changed, we update the local files. On start-up, we attempt to read from the pickle file; if the file doesn’t exist, we do a basic initialization and create the pickle files.