“Distributed Calendar Implementing Wuu and Bernstein’s Algorithm”

Nick Marton and Ian O’Boyle

Our implementation makes use of three objects: Node objects, Event objects and Appointment objects. The classes for these objects are implemented in Node.py, Event.py, and Appointment.py respectively.

An Appointment object contains a name, a date, a start time, an end time, and the names of all other users associated with the appointment. An Event, stored in a Node’s log, represents an action that takes place at a Node. An Event includes parameters for the Event’s operation type (and parameters of that operation), the ID number of the Node that created the Event object, and a time-stamp of the Event’s creation at that Node. Event and Appointment objects are never used directly and instead are used by the Node class to implement the algorithm. The Node class is implemented in Node.py. A Node object has a unique ID, a local clock of the Node enforced as an integer and incremented whenever it is referenced, a local calendar of events maintained as a dictionary data structure (with Appointment names as the keys and the Appointment objects themselves as the values), a local log of event records maintained by this Node, a 2D Time Table of Integers, the number of total Nodes in the distributed system enforced as an integer, and a dictionary of form [Int: (IPADDRESS, PORT\_NUMBER)], containing the Node IDs to IP address translations of the system.

To start running a node instance on the command line, issue the following command: python Node.py [NODE ID] [PORT NUMBER]. To schedule or cancel an event, use a command of the form:

“acting\_user {schedules, cancels} name\_of\_appointment (tuple\_of\_users\_involved) (start\_time, end\_time) day\_of\_week”.

When a user schedules an event, their request is parsed into an Appointment object and the Node checks it's local dictionary for conflicts before insertion. In the case of no conflicts, insertion happens normally as described in the Wuu-Bernstein algorithm.

In the case of a conflicting appointment, the Node saves a copy of its dictionary state before receiving (implicitly updating its knowledge from the message). After receive is finished, the Node will extract the events corresponding to new insertions of Appointments that were not present in the pre-receive dictionary and are present in the dictionary post-receive.

For those new events ( and corresponding new Appointments) that conflict with the dictionary after receive, the Node finds the event it conflicts with and the corresponding Appointment, which we call the *original event* and *original appointment* respectively. We consider two cases with our conflict resolution: the case where the original event is in the Node's log and the case where it is not. In first case, we delete the new Appointment from the dictionary as the Node currently resolving the conflict knows that all Nodes know of events up until the time of the original event and therefore all node's are in agreement for the original event. In the second case where the Node's log contains the original event, we discard the Appointment corresponding to the Event with a greater time-stamp in the Node's table on the Node's row (i.e, if Ti[i][oR.node] < Ti[i][nR.node] where i is the id of this node and oR an nR are the event records denoting the original event and new event respectively, we discard the Appointment corresponding to the new event nR) effectively choosing to keep the event that happened earlier.