DSA Project 2

Mitesh Kumar (kumarm4), Jason Lam (lamj7)

Project Design and Implementation

Data structures: Each site maintains a set of data structures in order to participate in the algorithm. The dictionary is stored as a hashmap from the order name to the order object. The order object contains the amounts of the order, as well as it's current status. The partial log is stored as an ordered list of log entries. Each log entry stores an op-code, order name, order amounts, sitename of the site it was executed on, and a vector timestamp of the site it was executed on, at the time of execution. The partial log is maintained such that the log entries are ordered first by the happens before relationship, that we can deduce from the vector clock stored with each log entry, and then if the two entries are concurrent, by lexicographical order by their names. The matrix clock is stored as a nested hash table, where the value at key i and secondary key j represent the matrix clock value between site i and site j. Within each message, we store a partial log, a matrix clock, and the sender's site id.

Stable storage implementation: We serialize all of our data structures and store them on disk after every modification to state, and we always try to initialize all of our data structures from the serialized data before creating them from scratch. In this way, all operations will always be atomic. If we crash mid way through an operation, then upon start up, it will be as if the operation never happened. Also, once we commit the operation, we will not lose that progress of state if we crash and then restart.

Use of sockets: We implemented a select based single threaded server with event handlers for each of the UI components, as well as a handler for the receive of a message. We utilize the select statement in Go in order to wait for standard input and network UDP input simultaneously. In order to receive incoming messages, we bind a UDP socket endpoint to the first port available to the server, and then indefinitely loop and poll this endpoint for UDP datagrams from all addresses.

Use of threads: Our design is primarily single threaded. We only utilize two additional helper threads to poll standard input and the UDP socket since Go does not have support for the select system call like C. Instead, we have a dedicated stdin reader, and a dedicated UDP reader thread that each push their messages to a go channel. We can then use Go's select statement to somewhat mimic the behavior of the select system call in C.

Which Events Included In Each Message?

send <site_id>:

- When sending a message to another site, we send a partial log, our matrix clock, and our own site_id, so that the receiving site can know which site the message came from.
- The partial log is generated by applying the has_rec function to filter our own partial log. Has_rec checks whether we know if the recipient already has a particular log event. If it does, then we don't need to send it to them.
- We make sure to maintain the causal alphabetical order in this new filtered log.

Sendall:

- We apply the logic for send <site_id> individually to all site_ids.

When Can Pending Order Be Filled?

A pending order can be filled when

1. The fill condition has been checked on all causally preceding orders and they have either been filled, canceled, or left as pending
2. There is no corresponding cancel event (i.e. it exists in the dictionary)
3. It can be removed in log truncation because it has been seen by all other sites (i.e. hasRec is true for all sites)
4. There must be enough resources to fill the order if we were to fill all concurrent preceding orders which don't have a cancel order that causally precedes the pending order, where an order is defined to be a concurrent precedent of the pending order if it is concurrent with it, and has a lexicographically lower customer_name.

We must decide on orders in causal order, so we need condition 1. If there is a corresponding cancel event, then the order should have been removed from the dictionary already, which is why we need condition 2. If we don't know whether an order has been received on all sites, then we can't be certain that we have all concurrent precedents of the pending order in the local log which is why we need condition 3. We want all sites to agree on a total ordering of the concurrent events, which is why we need condition 4.

Suppose we have the following events:
- Insert X
- Insert A
- Insert Y

Where, (Insert X) => (Insert A), (Insert X) || (Insert Y), (Insert A) || (Insert Y), (Insert X) and (Insert Y) are in log Lj, (Insert A) is not in log Lj. Suppose site j knows that (Insert Y) is in the log of all other processes.

If (Insert X) and (Insert A) can both be filled, and (Insert X), (Insert Y) can both be filled, but (Insert X), (Insert A), (Insert Y) cannot all three be filled, we have to show that (Insert A) cannot possibly be filled before (Insert Y) on any node. This would be sufficient to show that (Insert Y) can safely be filled as long as (Insert X) and (Insert Y) can be safely filled, where (Insert X) is a concurrent precedent of (Insert Y). In order for (Insert A) to be filled at process i, process i at some point has to send its log containing (Insert A) to process j. Process j at some point has to send its log containing (Insert Y) to process i, and potentially a (cancel A) if (insert X) meets the fill condition. In all cases, (Insert A)'s fill condition will always be checked after (Insert Y) has already been processed, which shows that we only have to check (Insert Y)'s causal precedents that are in the actual log Lj in order to safely say that we can fill it.

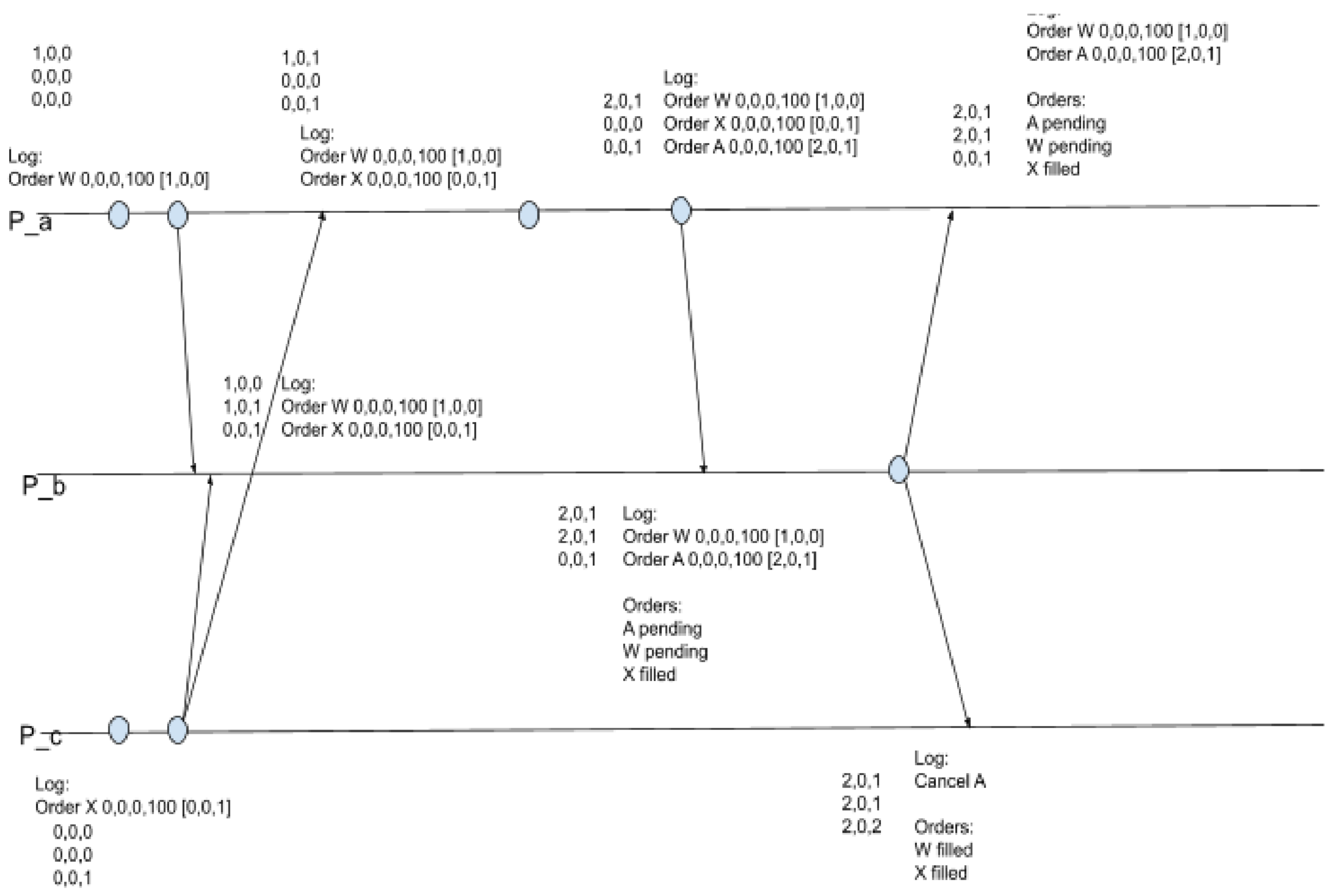
Testing

- Crashes were tested by using the quit command and then restarting the program and making sure that the data structures have remained in the same state.
- Message loss was tested by not sending a message. Message sends do not result in any state change on the sender, so if the message is lost, there is no effect on the correctness of the algorithm.

Team Work

Time spent: 20 hours total amongst team members

- Mitesh: Developed working application of assignment, testing, comments, and report
- Jason: Helped with testing, and report


1,0,0
0,0,0
0,0,0
Log:
Order W 0,0,0,100 [1,0,0]
1,0,1
0,0,0
0,0,1
Log:
Order W 0,0,0,100 [1,0,0]
Order X 0,0,0,100 [0,0,1]
2,0,1
0,0,0
0,0,1
Log:
Order W 0,0,0,100 [1,0,0]
Order X 0,0,0,100 [0,0,1]
Order A 0,0,0,100 [2,0,1]
Order W 0,0,0,100 [1,0,0]
Order A 0,0,0,100 [2,0,1]
2,0,1
2,0,1
0,0,1
Orders:
A pending
W pending
X filled
P_a
1,0,0
1,0,1
0,0,1
Log:
Order W 0,0,0,100 [1,0,0]
Order X 0,0,0,100 [0,0,1]
P_b
2,0,1
2,0,1
0,0,1
Log:
Order W 0,0,0,100 [1,0,0]
Order A 0,0,0,100 [2,0,1]
Orders:
A pending
W pending
X filled
P_c
Log:
Order X 0,0,0,100 [0,0,1]
0,0,0
0,0,0
0,0,1
2,0,1
2,0,1
2,0,2
Log:
Cancel A
Orders:
W filled
X filled

This test verifies the behavior of the algorithm when we have two nodes that have different orders which are truncatable, but both cannot be applied simultaneously. We can observe that the causal precedent checks allow P_a to not fill order A, which would cause a conflict with order W and order X that are filled on P_C.