**CSE 490h – Project 4 – PAXOS**

**Implementation Details**

We made several changes to our existing transaction and RIO systems in addition to implementing the PAXOS algorithm for committing changes for this project. Changes to the existing functionality provided by project 3 are outlined in the **Fixes of Known Issues** section. The addition of the PAXOS algorithm is outlined by the **PAXOS** section. System file formats and their purposes are partially introduced in the **Fixes of Known Issues** and **PAXOS** sections and outlined specifically in the **System Files** section. How timeouts in the transaction layer are handled is outlined in the **Transaction Layer Timeouts** section.

**Fixes of Known Issues**

We made a change to the RIO layer that stores session IDs on disk. We realized that not doing this did not guarantee at-most-once semantics after reading the feedback from project 1. These are stored in the “.sessions” file on disk.

We also fixed some known issues in our transaction layer for this project. We were not handling the case if a server crashed in the middle of writing a commit to disk. Now we use the file “.wh\_log”, and create a redo/write-ahead log for all updates that will be performed on the files that the transaction uses. If this file exists on startup, we execute the file updates in it. The updates contained in the file are not every command in the transaction, but rather the final state of each file used by the transaction at the time the transaction commits.

Previously, we used the “.l” file to store filenames present on the system on the server. We were not storing the file version though, which presented a problem if the server crashed, restarted, and then received a commit which had dependencies on a previously committed version of the file. We added the last committed file version to this file, so that when the server starts up, it loads each filename and its last committed version.

One other previous problem mentioned in project 3’s write-up is that there weren’t timeouts in the transaction layer. So after a client’s commit was ACKed, it would have no way of recovering if the server crashed except by restarting. For this project, timeouts are implemented by the class TimeoutManager in the transaction layer. It creates timeouts similar to the RIO layer, and stores the last sequence numbers sent for each node it communicates with. For clients, timeouts are created for the following TXNProtocols: WQ, CREATE, ABORT, COMMIT, START. For servers, timeouts are created for the following TXNProtocols: WF. How each timeout is handled is described more in the timeouts section.

We were not using transaction ids to identify transactions in the previous project, which presents a problem if a client tries to commit a transaction twice. This also complicates things when determining whether not tx2 can commit if it is dependent on tx1, and whether or not tx1 was the last committed transaction for the dependent file. So we changed the identifier for a transaction to be a unique transaction id as opposed to the committer’s address. Each transaction id starts out as a given node’s address, and then is incremented by RIONode.NUM\_NODES. This way all transaction ids are unique and the node’s address can be pulled from the transaction id by modding by the number of nodes. Each client writes their last committed transaction id to disk (stored on clients in file “.txn\_id”), so that if they crash and then they start up, they will start with a new, unique transaction id. The server takes advantage of using transaction ids by storing all transaction ids and whether they committed or aborted. This is also stored on disk, in file “.txn\_log”.

Another problem fixed for this project relates to tracking dependencies of a transaction. Previously, where a client got a file from was stored in memory on the server, so if the server crashed, the server would have the client abort in the best case, or allow an inconsistent commit in the worst case. This was fixed by storing where a client got a file from on the client side. When a client gets a write data back from the server, the packet contains what transaction the file came from (or 0 if it was from the server/last committed version). The client stores this and passes it to the server when it commits.

We changed much of how the commit process works in addition to adding PAXOS. Previously we were sending every commit as a sequence of packets, but had no way of recovering if either node crashed during this process. If the client crashed and came back up, then tried to commit, the new commit packets would be included with the old ones, which is another reason we changed to transaction ids. We changed this process to sending only one commit packet, which contained all the commands in the transaction. We are assuming that all transactions fit into one packet for simplifying purposes. We use the CommitPacket for this purpose, which is wrapped by a TXNPacket before being handed off to the RIO layer. The CommitPacket builds a string representation of the transaction by using the buildCommit function in the Transaction class. Contents of put and append commands are transformed into byte arrays before being put in the transaction string to prevent conflicts with using delimiters in the transaction string.

One additional thing we failed to mention in our previous write-ups is that you can send content to the put and append commands with whitespace in them, but then the contents must be surrounded by double quotes.

Other than these changes, and the addition to the PAXOS algorithm, the functionality is similar to what was described in the project 3 write-up.

**PAXOS**

**Transaction Layer Timeouts**

**Server Timeouts:**

* WF – Write Forward
  + When a write forward times out, the server removes that node from the nodes it is waiting on. Then the server checks to see if it waiting on any other writes forwards. If it isn’t, then it will respond to the client who requested the file with the highest version of the file it knows about. Otherwise it waits for all other write forwards to either return or timeout.

**Client Timeouts:**

* WQ – Write Query, CREATE
  + When a write query times out, an error message is printed out, and the command is not executed. If the server didn’t crash, and was just delayed, and a write data gets returned, it is ignored by the client, since it will have the sequence number of the write query that was originally sent, which timed out.
* COMMIT, ABORT
  + When a commit or abort time out, the client prints out an error message saying so. If the server didn’t crash but is just taking a long time, and returns a confirmation of a commit or abort, the client still executes the transaction and prints out a success message. -> should change to infinite retries?
* START
  + When a start times out, the client prints out an error message, and user must type the command again.

**System Files**