Andrew Koulogeorge
440 P4 Writeup (last one; lfg)

Protocol between Server and UserNodes

Serialization: **I implemented a Seralizer class for both the Server and UserNode** class to handle communication over the network. Each packet contains a Header which stores an Op code representing the type of message being sent as well as the unique transaction ID. Based on the header information, data stored in the body is parsed.

Asynchronous Communication: All of the communication between the Coordinator and the User Nodes is done **asynchronously with the deliverMessage function**. One benefit of this design choice was that on the server side, Asynchronous communication avoids issues such as one thread pulling a message from the queue that was intended for a different thread since there is a shared queue among the entire Coordinator process. To resolve this, I updated global thread safe hashmaps that store information about the incoming messages. These same data structures are then queried within while loops to see if all messages have arrived (both in the vote and ack collection stage)

Timeout Thresholds and Lost Messages

The standard 2PC implementation does not guarantee protocol termination due to the possibility of indefinite, well timed actions from Murphy. To prevent this, I implemented Timeouts during both the Vote Gathering and Vote Distribution phase of the 2PC protocol.

Stage 1: Vote Gathering: After sending out vote requests from each of the User Nodes involved in the transaction, the Coordinator enters in a loop for 3 seconds to collect the answers of the User Nodes. After 3 seconds is over, if there are still User Nodes which have not sent their Vote for the transaction, the Coordinator aborts the transaction. I chose **3 seconds** since the write up says that we can assume that any message not received within 3 seconds can be assumed to be lost.

Stage 2: Vote Distribution After sending out a commit message to all the User Nodes involved in the transaction, the Coordinator must be more stubborn than it was in Stage 1 since we need to ensure that each of the User Nodes committed their state to prevent there being inconsistencies across the Coordinator and all the User Nodes. Every 3 seconds I check which User Nodes have not sent an Ack to the Coordinator and I resend the commit message to those User Nodes. I repeat this **retransmission** 20 times before reporting a failure message and terminating the transaction. This results in a total timeout threshold of around 60 seconds. **Note that in this fault case, if a node permanently dies after voting yes but before local commitment, we cannot guarantee correct behavior no matter how long we send retransmissions.**

Recover from Node Failures

Coordinator Logging: My design for recovery handling is motivated by the following 3 cases when a server process is terminated in the middle of the 2PC protocol:

1) Termination occurred before commitment occurred on the server side
2) Termination occurred after commitment by the Server but before all acks were collected from User Nodes
3) Termination occurred after all acks were collected from User Nodes

My writing to logs occurs in 3 places during a transaction corresponding to these 3 events. Before a vote gather is called by the Coordinator, I save the list of users involved in the transaction. When the Coordinator gets back all yes votes from the User Nodes, I save the outcome and the filename where the collage will be saved. In the case where the outcome of the vote is Commit, I also write the img byte array to stable storage. Note that we need to write the img byte array to stable storage before writing it to memory since the server could crash in between the log and the write to memory step, resulting in the image being lost. After all the acks are collected by the Coordinator, I save in the log that this transaction completed. In the log file, each writes is marked with a corresponding Opcode to indicate the log type.

Coordinator Recovery: In a failure, I traverse over the logs line by line and keep a data structure that maps transaction id → most recent log type performed. Since information is written to the log in sequential order, this data structure allows me to identify where each transaction was in its lifecycle when the crash occurred. Based on which of the 3 above cases the transaction was in, I perform the following recovery logic:

1) Send ABORT to each of the User Nodes to cancel the transaction. The list of User Nodes from this transaction was saved in the first logging action taken
2) Send COMMIT to each of the User Nodes and then block to collect the acks.
3) Do nothing! The transaction completed successfully.

UserNode Logging: Again, my recovery handling design is motivated by the following 2 cases when a user node process is terminated during the 2PC protocol:

1) Termination occurs after the User Node agrees to commitment but before it sends an ack to the Coordinator
2) Termination occurs after ack is sent to the coordinator

UserNode Recovery:

1) Lock all files used in the failed transaction and send the Coordinator a message indicating its yes vote. The User Node then waits for the Coordinator to resend its commit message and will proceed as usual
2) Do nothing. Transaction completed.

Other Notes

I include an **EOL string** at the end of each of my log entries to mark that the log was successfully written to. The writing of the string to the file is not an atomic operation and, in a real system, **if a crash occurred during writing to the log you would want to be able to correctly identify and handle this case**. As a result, while I am traversing log entries, I skip over any entry that does not have an EOL string at the end.

All of the operations on the User Node side such as deleteFiles and removeStagingGround were implemented specifically to **not have any effect if called more then once**. This enables correct behavior in the face of aggressive failures. For example, there could be a case where the User Node committed locally but crashed before it was able to write to a stable stage. In this case the node would commit its state again on recovery and my implementation covers this.