PART TWO

Joshua Gu

ID: 34466505

To implement totally ordered writes across multiple instances of the replicated DB servers, I used the totally ordered multicast protocol. This consists of three stages. In stage 1, a server receives a request from the client. The server then multicasts an UPDATE message to all servers including itself. The server also places the client address in a hashmap to track the number of times the server owes a response to that client. The server does not respond to the client yet. In stage 2, when any server receives an UPDATE message, the server adds the message to its priority queue sorted by lamport clocks along with a server ID tiebreaker. The server then multicasts an ACK message to all servers including itself. In stage 3, whenever a server receives an ACK message, it updates a counter in a hashmap for the associated message. It also updates its lamport clock. Once the front of the priority queue receives ACKS from all servers indicated by a counter, the server delivers the message and removes it from the priority queue. If the server also owes the client a message, the server sends a response to the client.

Messages for hashmap and priority queue storage use a similar encoding format as a string. Both data structures encode the message as "<command>|<callback_id>|<client_address>|<server_id>|<message_lamport>", where server_id and message_lamport are used for sorting within the priority queue. Callback_id is used for callbacks in the MyDBClient, but if the server receives no callback_id, it adds a callback_id of -1 so that messages stay consistent. To ensure message type identification and correct lamport clock updates, when a server multicasts a

message, it also appends its lamport clock and either “UPDATE” or “ACK” to the end of the message. The “|” symbol is used as a delimiter to split and decode the string.

PART ONE

Joshua Gu

ID: 34466505

To identify the right callbacks from the server response, I associated with each active callback an id called **callback_id**. This is a long variable that is increased by 1 each time **callbackSend** is called, ensuring that each callback has an associated unique id. The waiting callbacks are stored in the hashmap **callbacks** which map callback id's to callbacks. The hashmap and id's are protected under a lock, ensuring thread safety.

In MyDBClient.java, the callback id is appended to the end of the message with a "|" separator. The pipe symbol was chosen since the Cassandra commands do not use it, meaning it is safe to use as a delimiter. Since the server echos that message back, we can extract the callback id that is next to the delimiter and find its callback in the hashmap. On the server side, the server removes any characters to the right of and including the pipe symbol before relaying the message to the Cassandra instance. This means that the server can execute messages even if they do not use the callback encoding.