Concurrency Strategy

Overview

To deal with concurrency, we have created conversation objects that the server will hold and synchronize. The concurrency strategy that we have developed makes sure that race conditions do not occur within each conversation, and it also ensures that deadlock cannot occur. If a user sends the server a message, the server will synchronize a conversation object, post the command to the conversation object, and add the command to the queue of each of the users in the conversation. If a user logs out, logs in, or tracks another user, the user object will be synchronized. The rest of the document will explain the process in detail.

Implementation Details

First, all commands that a client sends to the server regarding changes in user objects will synchronize the user object being modified. The server keeps a list of all active user objects, and synchronization makes sure that only one thread can access a user at a time. For instance, the trackUser() method allows a client to track the login status of other users. In order for this to method to work correctly, we need to make sure a friend does not log out before the response of the trackUser() method is sent. Since the user is synchronized, the track command will finish before the user can logout, and since all the user commands such as trackuser, login, and logout are constant time and extremely fast, this does not pose a problem to the user. The implementation does prevent a trackUser() method from displaying something incorrect, however. Suppose the user was not synchronized when it was being modified. Then the trackUser() command could begin as the user was still online. The user could possibly logout during this time, but the trackUser() object would not be notified of this, and would return that the user is currently logged on.

Next, the synchronization of conversation objects and the subsequent use of object queues ensure that the history of each conversation is always displayed consistently and correctly. The server receives a command then identifies which conversation object should receive a response. Once a conversation is identified, the server synchronizes the object as it sends the response. Once the conversation has been synchronized, a server response is sent back to all of the users in a particular conversation. The response goes onto the user’s queue of outgoing commands. Finally, once the user thread is available, the user will go through and perform all the tasks on its queue in sequential order of when the tasks were received.

The synchronization of conversation objects makes sure that there are no race conditions. The synchronization prevents different responses from being sent to different users at the same time. The outgoing commands queue for each user makes sure that each user gets the commands in the same order as the other users (although the commands may not necessarily be performed at the same time). This makes sure that any conversation between two users will show up in exactly the same format for both users, even if the messages take longer to appear for one user than for the other. For instance, consider two users x and y sending a message through a chat at the same time. The server will receive one command first, let us say x’s command, and will synchronize the chat so that y’s command will block. Then, the server will send the server’s response to x’s command to both x and y so that both users receive a new command in their queue. Then the conversation object is released and the same process is performed for y’s command. At the end, users x and y have the same commands on their queue in the same order, and even if user x is still performing some operation (making x’s queue take longer to finish than y’s), the commands will be performed in the same order.

Synchronizing the conversation and then appending the server response to each user’s outgoing command queue makes sure all commands are received in the same order. Moreover, deadlocks will not occur because all commands that lock the user take constant time and will always finish. It is possible for a conversation to block and wait for a user’s lock to be released, but the user’s lock will always be released in a timely manner. This ensures that each command will eventually acquire the conversation and user locks so that the server can append its responses to the users’ queues.