Master M1 MOSIG – UGA

Principles of Operating Systems and Concurrent Programming

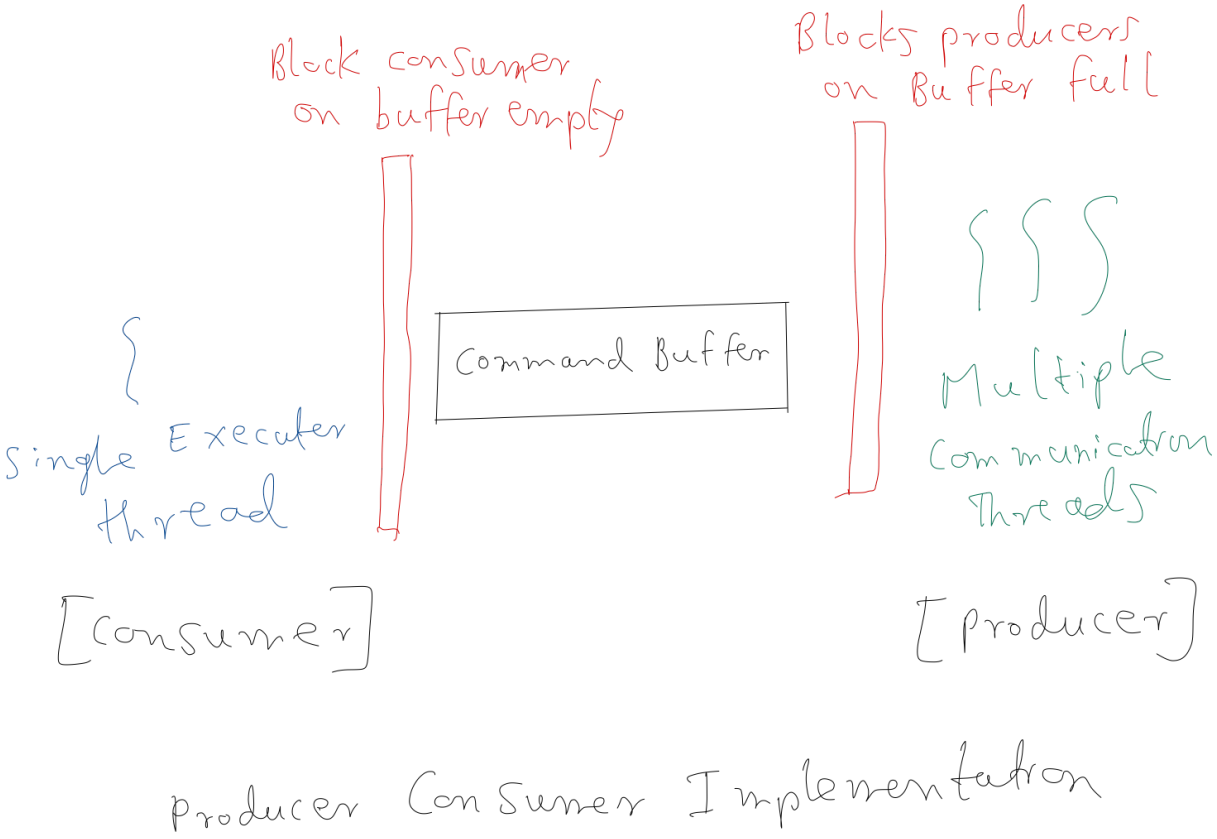
Babble – A multi-threaded server (for social networking) - Lab6 - Report

Author:
Jit Chatterjee
Arthur Chardon

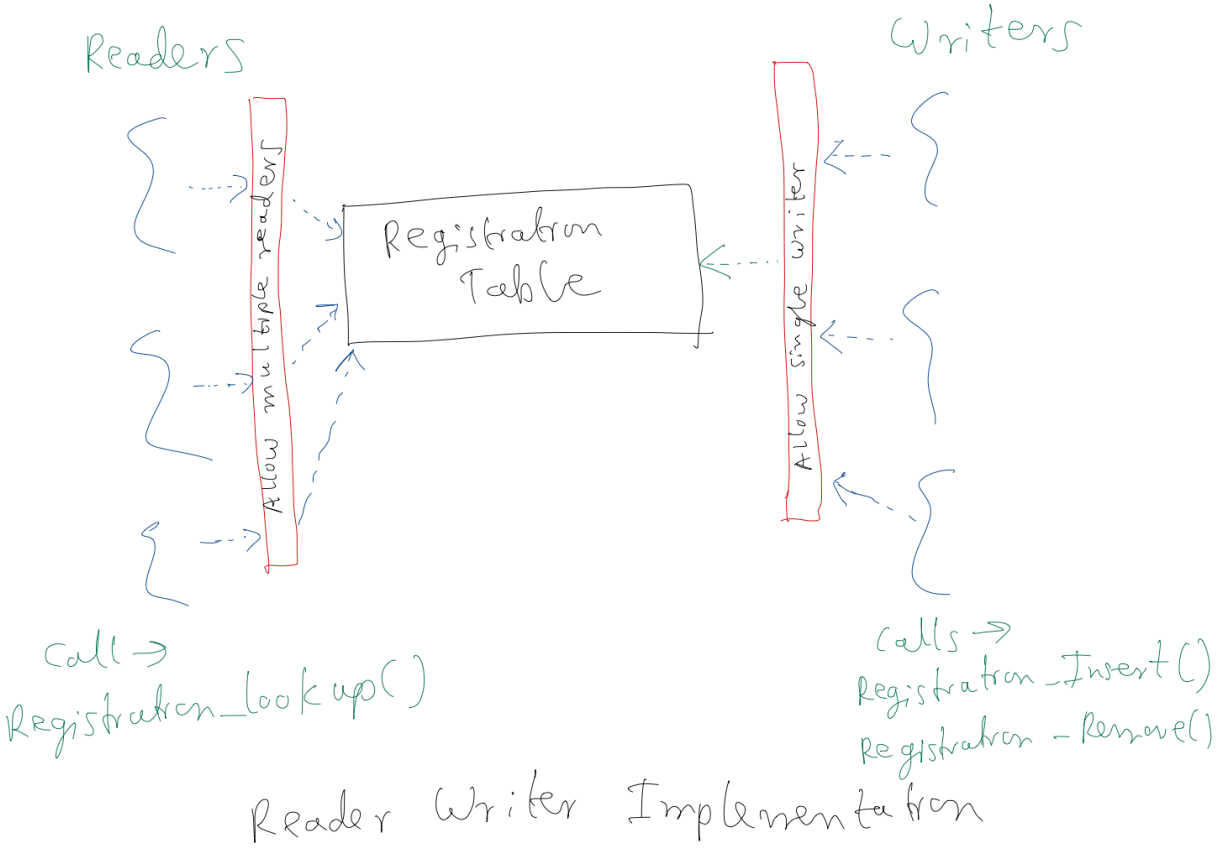
Submitted to:
Thomas Ropars
Renaud Lachaize

Stage 1:
In this stage there are multiple communication threads and a single executor thread. Therefore we implemented the synchronization with semaphores, to restrict the access to the critical section. We worked it as a producer-consumer problem on the server side for receiving and processing commands, and as a reader-writer problem when it comes to registration that also required synchronization.

The multiple communication threads take the client commands and put them into a command buffer. In this stage we have a single executor thread, to get the commands from the command buffer to execute the commands. So here we have implemented a producer-consumer using semaphores.



Moreover, there is a registration table where multiple threads are trying to read, write and insert into the table, so we have implemented a reader writer lock to solve this problem.



We have tested successfully with the follow_test and the stress_test with a large range of values. The performance_test results with a throughput of 43.50msg/s.

Stage 2:
In Stage 2, we have multiple executor threads. So now these multiple executor threads can create race conditions on the following commands PUBLISH, FOLLOW, TIMELINE. So we have to find out the critical section where we can put locks, so that we can eliminate race conditions.

Stage 2a:
In this stage, there are multiple execution threads along with the multiple communication threads. We decided to use locks, one for each command (FOLLOW, PUBLISH, TIMELINE) to protect the critical section. The locks are protecting the variable *nb_followers* which is a data type of the structure *client_bundle*.

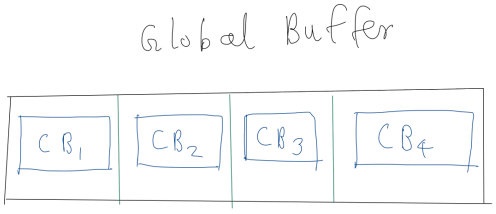
We also have tested successfully with the follow_test and the stress_test with a large range of values. The performance_test results also with a throughput of 43.50msg/s.

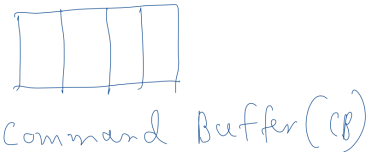
Stage 2b:
In this stage, a new problem arises when we use the RDV command. For each client, if the executor threads do not complete processing other commands like (FOLLOW, PUBLISH) and calls RDV command before that for that particular client. It will lead to a failure. So, we

have to implement this stage in a way so that we can get the timestamp of the commands for a particular client and process RDV after the completion of commands like (FOLLOW, PUBLISH) by the executor thread for that particular client. RDV is like an acknowledgment. So, after completion of the other commands for a particular client, the executor thread should call RDV.

So for this stage we thought of implementing a wait function which will wait till the other commands like (FOLLOW, PUBLISH) get completed and after that it will execute RDV for that particular client.

Stage 3:
In this stage we thought of increasing the performance by creating multiple command buffers. So to implement this we have to create an array of arrays. So the main idea is the multiple command buffers will be stored inside an array (let suppose named as Global_Buffer) and this buffer also needs to be protected using producer consumer. So, the multiple command buffers are already being protected by producer consumer and this Global_Buffer is also being protected by another producer consumer.





Feedback:
We have learned a lot regarding different synchronization techniques but the main challenge was to test it because it randomly fails. So, we have created scripts which will run the test many times so that we can figure out if it's working or not.