Lab 7: Babble - A multi-threaded server
Group Report

Do Son Tung, Mariia Zameshina

Summary

In this lab stages $1, 2, 3, 4$ were completed and tested.

1 Stage 1: Accepting multiple client connections

To accept multiple clients, in the main server thread, we repeatedly accept new client. Each client we accept correspond to a new communication thread to be created.

Since we have multiple communication threads, We implemented executor threads to process and execute command from all the clients.

Producer-consumer problem

First of all, we have a **producer-consumer problem**: communication thread puts elements in command buffer and executor thread takes elements from command buffer. We implemented

- **void* put_in_command_buffer(command_t* cmd)**
- **command_t* get_from_command_buffer()**

for communication thread and executor thread respectively using Mutexes and Conditional variables. This solution will solve the synchronization in command buffer.

Reader-writer problem

Secondly, for the **reader-writer problem**: we used rwlock.

```
pthread_rwlock_wrlock(&rwlock)
pthread_rwlock_rdlock(&rwlock)
pthread_rwlock_unlock(&rwlock)
```

With this lock, we apply rdlock to lookup function in registration so it allows multiple readers (read-only) and we apply wrlock to insert, remove functions in registration so it allows at most one writer (exclusively).

Tests

The solution was checked on the following tests:

- **Follow test:** Tests which are passed:

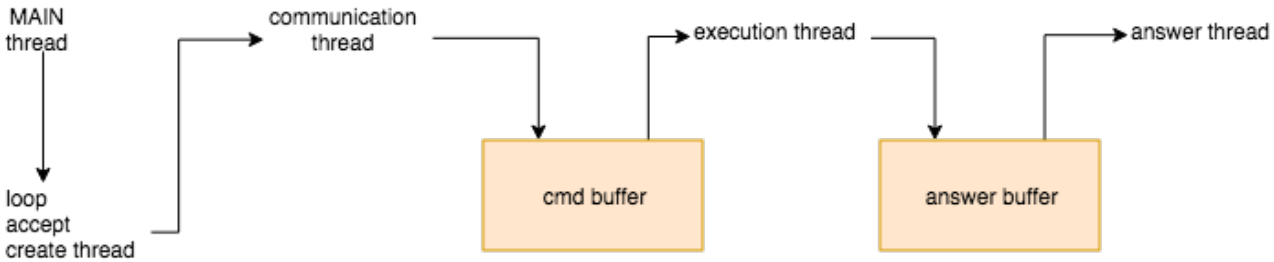
```
./follow_test.run -t 10;
./follow_test.run -t 100;
./follow_test.run -t 1000;
```

- **Stress test:** Tests which are passed:

```
./stress_test.run -n 1 -k 1000;
./stress_test.run -n 1000 -k 10;
./stress_test.run -n 100 -k 100;
./stress_test.run -n 300 -k 1000;
./stress_test.run -n 500 -k 800;
./stress_test.run -n 67 -k 10000 (took about 1hr to execute);
```

2 Stage 2: Threads dedicated to answer clients

2.1 Producer-consumer problem



In order to implement the answer threads, first of all, the answer buffer was created. It is used by answer threads in the similar way to use of command buffer (and again we see the **producer-consumer problem**) : we create 2 new functions

- **void* put_in_answer_buffer(answer_t* ans)**
- **answer_t* get_from_answer_buffer()**

which are implemented in executor threads and answer threads respectively. This solution uses mutexes and conditional variables as well.

Answer thread gets answer from the answer buffer (using the implemented function) and sends it to client.

3 Stage 3: Running multiple executor threads

In this stage instead of one executor thread, at the start of server BABBLE_ANSWER_THREADS of executor threads were created. We detected two problems:

- In FOLLOW and PUBLISH command, we access to a client_bundle information and modify it outside registration table functions provided. This means there is a shared data can be modified by multiple executor threads.
  Solution: We provided mutex lock in this two specific commands to prevent this problem.

- There is a problem with RDV command, RDV should be executed after all the request from a client has been processed. But due to multiple executor threads, it can lead to the situation where some RDV commands are executed before others command.
  Solution: We have a new attribute in client_bundle to count the number of remaining command that need to be executed. If a executor thread get a RDV command, it will look up at the counter. If there are remaining commands, it will be blocked. (N.B: We only count the command that not RDV)

Tests

- **Follow test** Tests which are passed:

```
./follow_test.run -t 10;
./follow_test.run -t 100;
```

- **Stress test** tests which are passed:

```
./stress_test.run -n 1 -k 10;
./stress_test.run -n 100 -k 100;
./stress_test.run -n 300 -k 300;
./stress_test.run -n 20 -k 200;
./stress_test.run -n 1000 -k 100;
./stress_test.run -n 1000 -k 1000;
```

4 Stage 4: Processing publish actions with high priority

In this stage, what we want to do it to looking for the PUBLISH command every time a executor thread try to get a new command from the buffer, it will re-arrange the buffer so that one PUBLISH command come first. Re-arrange function is called in function *command_t* get_from_command_buffer()*.

Out is the index for executor thread as the next item to be processed and *In* is the index for communication threads as the index of the next slot to put item in, and *index* is the slot where you found a publish command.

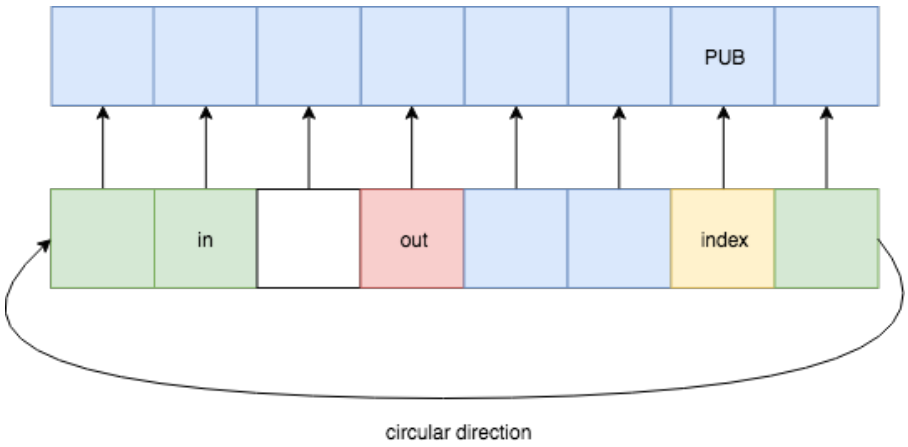


Fig. 1: *Before state*

The idea is to place the command in *index* slot to *out* slot, and push every thing else 1 slot to the right. Command buffer is a buffer of pointers to command. What we did are change the value which they pointed to. We do it in the loop and iterative backward from the index point. After we re-arrange it, the executor will still pick up the command at *out* index but the content is what we want now. You can see the state before and after in Fig. 1, and 2

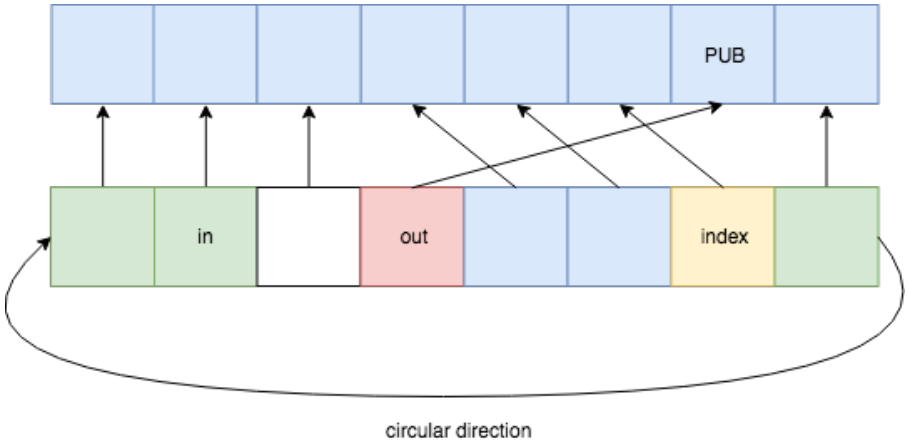


Fig. 2: *After state*

Tests

For the testing stage 4 one can use sleep before pushing the commands to cmd buffer by length equal to some fixed constant minus (start time minus time of appearance of the command). After that one can just simulate the order of commands FOLLOW and PUBLISH in console for couple of clients and see if commands are executed in the right order.

5 Stage 5: Dealing with celebrities

Not yet implemented.

Conclusion

We successfully implemented stage 1,2,3, and 4. All the stress test are passed. The number of client we tested is up to 1000 clients with 10 to 100s messages.

All the stage ensure the safety property of the synchronization problems. Only one remark on the performance is when the number of client is increase, we observe that the time that we wait after the registration successful is high with the stress test.