Principles of Operating Systems and Concurrent Programming

Babble - Multi-Threaded Server

Submitted By:
Bushra Aboubakr, Salman Farhat

Contents

1 Stage 1 - multiple client connections **2**
1.1 Concurrency issues 2
1.2 Designed solution 2

2 Stage 2 - answer threads **3**
2.1 Concurrency issues 3
2.2 Designed solution 3

3 Stage 3 - multiple executor threads **3**
3.1 Concurrency issues 3
3.2 Designed solution 4

4 Stage 4 - priority **4**
4.1 Designed solution 4

5 Tests **4**

1 Stage 1 - multiple client connections

1.1 Concurrency issues

1. Producer-consumer problem in the command_buffer.
As multiple communicators, and an executor will use command_buffer to read/write the commands. That will lead to the in-consistency state in the buffer.

2. Read-Write problem in the registration table.
As multiple threads will look-up, insert, or remove entries (clients) from the registration table simultaneously.

3. Rendezvous function issue. The server might send rendezvous acknowledgment to client before FOLLOW and PUBLISH commands finished processing.

1.2 Designed solution

Identify the critical sections, where the command buffer and registration table are modifying, in addition to babble_implementation functions.

In all stages, we concerns on the part the can be accessed simultaneously (the global variables), and the concurrent access as well.

The classes modified are (babble_server, babble_implementation , babble_registration, babble_timeline and babble_types).

Command Buffer: In the bubble_server, the critical sections of reading/writing process from the command buffer are protected.

Registration Table: in the babble_registration class: a pthread rwlocks used instead of semaphores or conditional variables as we wanted to not give a priority to reader or writer against the other, in specifically we want to avoid the starvation to both.

In the lookup function, we used a read lock, while in insert and remove a write lock used around the critical sections, which includes the code that is using the global variable "nb_registered_clients", and the writing process in the registration table.

Run rdv command: count the number of FOLLOW and PUBLISH commands that still under-processing, and hold the rdv-ack until this counter is equal to zero.

This counter needs to be protected in case of multiple executors case (in stage 3).

2 Stage 2 - answer threads

2.1 Concurrency issues

1. Producer-consumer problem in the answer_buffer.
As multiple answer threads will read from the buffer, and an executor will write in it.
2. Multiple publish commands may be processed concurrently for the same client and so modified the timeline at the same time.

2.2 Designed solution

1. In the bubble_server, protect the critical sections in the executor_routine and answer_routine.

2. protect the insert and generate summery functions in the babble_timeline.

3 Stage 3 - multiple executor threads

3.1 Concurrency issues

Commands run functions in server_implementation need to be protected as they will be executed simultaneously.

1. run_publish_command: the number of followers will be updated by multiple threads. The same problem for the called function timeline_insert.
2. run_follow_command: the number of followers will be updated by multiple threads.
3. run_timeline_command: it calls a timeline_generate_summary function which updates two global variables count_recent_adds and circular_buffer.

3.2 Designed solution

(1)(2) Protect the mentioned variables and functions with mutex.
(3) Use a rwlock, where the insert function needs a write lock, while the summery generation needs a read lock.

4 Stage 4 - priority

4.1 Designed solution

add another buffer for the publish commands to handle the priority in separate. This solution give more dynamic (and most probably easiest) if we need to add more priority constraints later.

On the other hand, we limit the number of commands space in the buffer for the commands other than the publish commands, for example, if a space for commands buffer is n, so we divides it between the publish commands and the rest of the commands. That might not cause an impact as we have a smaller number of executor than the buffer size.

The hypothesis made:
In case the publish buffer is full, we will wait until there will be some empty space. As if we added the publish buffer to the command buffer, there is no guarantee that this command will be processed before if we let the communicator wait until the executor some publish commands.

5 Tests

1. follow test (+s) passed.
2. stress test (+s) passed. It is tested on the extensive number of clients, and a few number of clients and extensive messages. (up to 1000)
3. tested with 3 and 5 executors.