Lab 7: Babble – A multi-threaded server (for social networking)

Master M1 MOSIG – Université Grenoble Alpes & Grenoble INP

2017

In this lab, you are going to build a *real* multi-threaded application out of a sequential program. This program is rather complex, as it implements a client-server communication protocol. Hence, this lab is not only a chance to practice concurrent programming but also to get used to work with code of significant size.

During the different stages of the project, you will have to identify the concurrency issues related to the use of multiple threads, and to propose appropriate solutions to handle these issues.

1 Important information

- This assignment will be graded.
- The assignment is to be done by groups of at most **2** students.
- Deadline: **November, 27, 2017**.
- The assignment is to be turned in on Moodle.

1.1 Collaboration and plagiarism

You are encouraged to discuss ideas and problems related to this project with the other students. You can also look for additional resources on the Internet. However, we consider plagiarism very seriously. Hence, if any part of your final submission reflects influences from external sources, you must cite these external sources in your report. Also, any part of your design, your implementation, and your report should come from you and not from other students. We will run tests to detect similarities between source codes. Additionally, we will allow ourselves to question you about your submission if part of it looks suspicious, which means that you should be able to explain each line you submitted. In case of plagiarism, your submission will not be graded.

1.2 Evaluation of your work

The main points that will be evaluated for this work are (ordered by priority):

- The understanding of the problem.
- The design of the proposed solution.
- The correctness of the code.
- The quality of the code.
- The number of implemented features.

1.3 Your submission

Your submission is to be turned in on Moodle as an archive named with the last name of the two students involved in the project: `Name1_Name2_lab7.tar.gz`.

The archive will include:

- A **concise** report (either in `txt` or `pdf` format[1]) that should include the following sections:
  - The name of the participants.
  - For each stage of the project:
    - The answer to the questions raised in this document (when applicable).
    - A short description of each concurrency issue you have identified.
    - A few words about the solution you designed to solve it.
    - Any additional information you consider mandatory to understand your code.
    - A list of tests that you pass successfully and the known bugs if any.
    - A brief description of the new tests you have designed, if any (bonus).
  - A feedback section including any suggestions you would have to improve this lab.
- A version of your code for each stage of the project (except for stage 0), in a directory `stage_X` for stage X. Each version should be self-contained.

1.4 Expected achievements

Considering the time that is given to you to work on the assignment, here is a scale of our expectations:

- An **acceptable work** is one in which at least stage 1 has been implemented and works correctly.
- A **good work** is one in which stage 3 has been implemented, and works correctly.
- An **excellent work** is one in which, all stages have been implemented and extensively tested.

[1]Other formats will be rejected

1.5 About multi-threaded programming

In this lab, you are going to program with `pthreads`. To solve concurrency issues, you are allowed to use any synchronization mechanisms introduced during the lectures. This includes pthread mutexes, condition variables and semaphores. It also includes read-write locks (see `man pthread_rwlock_destroy`). Additionally, atomic operations can be used[2].

2 Overview

This lab is about the implementation of a client-server application providing a Twitter-like service. This service is called Babble. It allows users to register to the service and to publish messages. A user can also follow another user. The publications of a user are only visible to her followers.

2.1 About client-server communication

The goal of this lab is not to have you dealing with issues related to network communication. Hence, the code to communicate between Babble clients and the server is provided to you. In the following, we shortly describe network communication in C, for you to better understand the code you are going to manipulate. Many resources are available on the Internet (e.g., `http://pages.cs.wisc.edu/~suman/courses/640/lectures/sockets.pdf`) if you would like more information on the topic.

The principal OS abstraction for network communication is the socket. It is the point at which a process attaches to the network. In this lab, we are working with connection-oriented communication as it is the case for TCP. It means that a communication session is established between the client and the server before any application data is transferred. Here are the steps used to establish a connection between a server and a client (for more details, consult the manpages of the system calls):

1. A socket is identified by a port number. The port number is a value between 0 and 65535[3]. Babble uses port number 5656 by default.

Steps run on the server side:

2. A call to `socket()` is used to create a new socket on the server.
3. A call to `bind()` attaches the socket to the selected port.
4. Calling `listen()` informs that we are going to listen for new connections on this socket.
5. Calling `accept()` blocks until a new connection arrives on the socket. Then, it creates a new socket and return a file descriptor for that socket. This file descriptor will be used to read/write data to the connected client.

[2] A list of built-in atomic operations for gcc is available at `https://gcc.gnu.org/onlinedocs/gcc-4.4.0/gcc/Atomic-Builtins.html`

[3] Values from 0 to 1023 correspond to *system ports*.

Steps run on client-side:

6. On the client side, a socket should also first be created using `socket()`.
7. `connect()` is called to connect to the server socket defined by an IP address (or hostname) and the port number the server is listening on. If the client and the server are on the same host, `localhost` can be used as hostname (or IP address `127.0.0.1`). `connect()` also returns a file descriptor.

To communicate over the established connection, `read()` and `write()` operations can be run on the obtained file descriptors. The Babble connections are full-duplex: communication can happen in both directions at the same time.

2.2 The Babble messages

A set of messages are defined for the Babble communication protocol:

- `LOGIN id`: First message to send to register to the service. The `id` has a maximum size of 16 and can only include alphanumeric characters (no white spaces).
- `PUBLISH msg`: To publish a new message of max size 64. The message can include any characters but no white spaces.
- `FOLLOW id`: To follow another registered user. Note that at the time it registers, a user only follows herself.
- `TIMELINE`: To get a list of the messages published by the people the user follows since her last `TIMELINE` message. Note than when a user starts following another user, she does not get access to the previously published messages. Note also that if the timeline is big, an information about the total number of messages is sent, but the content of only the last 20 messages is included in the answer.
- `FOLLOW_COUNT`: To get the number of followers of the user.
- `RDV`: Used to check if all messages sent previously have been processed by the server.

It should be mentioned that in the current version of Babble, if a client disconnects and connects again later on with the same `id`, it will be considered as a new user. Also, the `FOLLOW` command only allows following users that are connected at the time the command is run.

2.3 Provided material

You are provided with a sequential implementation of Babble. It includes a Makefile to compile all the source files, and create the executable files for the server and the client (as well as some tests). Feel free to create additional source files to structure your code, but do not forget to update the Makefile accordingly.

The provided source files are:

- `babble_config.h`: Defines configuration parameters
- `babble_types.h`: Defines some of the main data structures of Babble
- `babble_publication_set*`: Operations on a set of publications
- `babble_registration*`: Management of data associated with connected clients
- `babble_communication*`: Utility functions to communicate between the server and the clients
- `babble_utils*`: Other utility functions
- `babble_server.h`: Defines the main functions used by the server.
- `babble_server_implem.c`: Implementation of the functions defined in `babble_server.h`
- `babble_client.h`: Defines the main functions used by the server.
- `babble_client_implem.c`: Implementation of the functions defined in `babble_client.h`
- `babble_server.c`: Implementation of the server
- `babble_client.c`: Implementation of the client
- `*_test.c`: Code of the provided tests

Important: Please do not hesitate to ask for explanations about the provided code by questioning the teaching staff during the lab sessions or through the forum on Moodle.

3 Stage 0: First contact with Babble

3.1 Running the code

It is now time to start working with Babble. As a very first step, we suggest you to try running the server and a client. After compiling the code, the following command can be used to run the server:

```
$ ./babble_server.run
```

To know the options that can be passed to the server executable, run with "`-h`".

In another terminal, launch the client:

```
$ ./babble_client.run -i my_id
```

Option "`-i`" is used to specify the identifier of the new client. Use "`-h`" to learn about other options.

Each client is identified on the server by a unique key which corresponds to a hash of its id. The client receives this key has an acknowledgment of successful login.

The babble client is a primitive console where you can type commands to be sent to the server. For instance, to publish the message "`voila`", simply type:

```
PUBLISH voila
```

Note that for the sake of simplicity, each command is also represented by a number (1 for PUBLISH) that can be used in the console. Hence, one can also enter:

```
1 voila
```

To find out what number correspond to each command, check `babble_types.h`.

Finally, to terminate the client console, one can use CTRL+D on an empty line to generate an *end-of-file* condition.

You can now play with Babble. However, you will soon notice that this sequential version is rather boring, as it accepts a single client connection at a time. But before starting implementing a multi-threaded version of Babble, let's try to better understand the code.

3.2 Digging into the code

To help you start getting familiar with the code of Babble, we suggest you to answer the following questions:

1. Which part of the code opens the socket where the server is going to listen for new connections?
2. Which part of the code manages new connections on server side?
3. What are the major steps that are run on the server when it receives a message from the client?
4. Why are LOGIN messages managed differently from other commands?
5. What is the purpose of the `registration_table`?
6. How are keys used on the server?
7. How answering to requests is implemented on the server?
8. What happens on the execution of a `FOLLOW` request?
9. On a TIMELINE request, how does the server ensures that it includes only new messages in the answer?
10. What are the high-level functions provided to communicate over the network? What is their return value?

4 Stage 1: Accepting multiple client connections

4.1 Evolution of the server

In this first stage, we would like to allow multiple clients to be registered to the service at the same time. To do so, we propose to create multiple threads on the server. Namely, we identify two new kinds of threads:

Communication threads. A communication thread is responsible for receiving from the network the messages sent by a client, and to parse these messages to generate commands to be executed.

Executor threads. An executor thread is responsible for executing pending commands and sending answers to clients.

In stage 1, a new communication thread should be created each time a new client connects to the server. On the other hand, a single executor thread will be created and will execute the commands of all clients.

To pass commands created by the communication threads to the executor thread, you should create a buffer (the `command_buffer`), and you should solve the producer-consumer synchronization issues on this buffer.

Figures 1, 2, and 3 provide a basic description of the sequence of tasks to be run by the threads in this stage. Note that all the code required to run these tasks is already provided to you: in the provided version, it is all executed by the main thread.

```
Parse input arguments
Initialize server data structures (call to server_data_init())
Initialize and open the server socket (call to server_connection_init())
while(true){
  Accept a new connexion and get a file descriptor on the new created socket (call to server_connection_accept())
  Start a new communication thread to handle the new client with the file descriptor as argument
}
```

Figure 1: Executed by the main server thread in Stage 1

```
Recv login message (call to network_recv())
Parse the message to get the command (call to parse_command())
Process the login command (call to process_command())
Send answer to the client (call to answer_command())
while( there are clients messages to process ){
  Parse the message to get the command
  Put the command in the command_buffer
}
Unregister the client (call to unregisted_client())
```

Figure 2: Executed by the communication thread in Stage 1

In addition to the producer-consumer problem previously mentioned, another issue is induced by these changes. Namely, there is one data structure that may be accessed concurrently by multiple threads. This is the `registration_table`. This table is read during the execution of multiple functions including `answer_command()` and it is updated during the processing of the login command as well as during the unregistration of the client.

```
while(true){
  Take a command from the command_buffer
  Process the command
  Send answer to the client
}
```

Figure 3: Executed by the executor thread in Stage 1

This creates a reader-writer problem that you should solve. We strongly recommand you to insert the synchronization code directly in the functions that implement access to the `registration_table`, that is in file `babble_registration.c`.

4.2 Testing the code

It is hard to put the server under stress condition by simply using the client console. This is why you are provided with two additional programs on client side to be used for testing :

follow_test: This test creates two threads (corresponding to two clients called `PUB` and `TIM`). `TIM` follows `PUB`. `PUB` continuously publishes new messages while `TIM` sends `TIMELINE` requests to get the new publications. At the end of the test, the number of publications received by `TIM` should correspond to the number of publications of `PUB`.

stress_test: This test creates several threads, each corresponding to a client. First, each client follows all other clients. At the end of the step, the test checks that the answers to `FOLLOW_COUNT` are consistent. Then, each thread publishes a sequence of messages. At the end of the second step, the test verifies that the answers to `TIMELINE` are consistent.

For each test, use "`-h`" to get information about the options. Do not hesitate to run the tests with different input parameters.

You can look at the code of the tests to see how they are implemented. You will notice the use of barriers (introduced in Lab 5) to synchronize threads before moving to a next step. Fill free to design your own testing programs if you think other testing scenarios can be interesting.

Important comment about tests: Testing multi-threaded applications is a very complex task. As multi-threading implies non-determinism, successfully running a test does not mean your program is correct. On the other hand, failing to run a test means that your program is not correct (assuming that the test itself is correct).

Streaming: The two provided tests have an option ("-s") to activate streaming. In streaming mode, the client does not wait for an answer for the commands that do not explicitly need it, that is, `FOLLOW` and `PUBLISH`. This allows stressing even more the server by having clients continuously sending new requests (to avoid totally overloading the server, in the provided test, there is a delay of 0.1 ms between two requests). Do not hesitate to test also with streaming activated.

Note that the use of streaming required introducing the `RDV` command for testing purpose. The `RDV` command asks for an acknowledgment by the server that is sent only when all the preceding requests of the client have been processed. Without this, it would be impossible for a client to know if all its `PUBLISH` requests have been handled in streaming mode.

Note also that the streaming mode is implemented by prefixing the request sent to the client with "`S`". As such, you can also test streaming in the console with:

```
S PUBLISH voila
```

5 Stage 2: Avoiding DoS attacks

Your new Babble application is very nice, as it enables multiple clients to register and to follow each other. However, your server is highly vulnerable to *Denial-of-Service* attacks: An attacker might keep creating new clients that connect to the server, so that, your server would spend all its time creating new communication threads instead of processing messages.

To avoid this problem, modify your server so that at most N communication threads[4] run at any time[5]. The idea is to create the N threads at the start of the server. Note that the main server thread should still be the only one calling `server_connection_accept()` to *accept* new clients' connections .

This modification implies that at most N clients can be registered to the service at the same time. This is a feature, not a bug.

6 Stage 3: Running multiple executor threads

6.1 Evolution of the server

Now that the server is safe, it is time to question its efficiency. It seems obvious that the server could be more efficient if there would be more than one executor thread[6].

In this stage, modify the server so that N executor threads are run[7].

In your report briefly explain the new issues related to concurrency that you identified, if any, and explain how you solved them.

6.2 Testing the new version

Test your new server with the provided test in normal execution mode. In a second step, test with streaming: having a correct solution for this case might be more subtle.

[4] `BABBLE_COMMUNICATION_THREADS` constant defined in `babble_config.h`

[5] This single change does not protect from DoS attacks, but such a discussion is outside the scope of this lab.

[6] You are not asked to show that it is actually (not?) more efficient.

[7] `BABBLE_EXECUTOR_THREADS` constant defined in `babble_config.h`.

7 Stage 4: Introducing multi-purposes threads

With this new version of the server, there is still a case where resources might not be used appropriately. Imagine that your Babble server is started with 5 communication threads and 2 executor threads, and that only two clients are connected. In this case, 3 communication threads will remain idle. Instead, we would like the communication threads to also process commands as long as there is no new client connection to handle: these *hybrid* threads are now both communication and executor threads.

In the report, briefly explain the new problems introduced by this evolution and your solutions.

Note: For this problem, using busy waiting will be accepted. Of course, avoiding busy waiting is still better.

Question: Can we replace all communication and executor threads and use only *hybrid* threads?

8 Stage 5 (bonus): Adding new features

If there is time left, please feel free to add any features that you would consider interesting to the Babble server, and explain them briefly in your report. Have fun!

We can suggest one direction for improvement related to client disconnection/re-connection. At this time, when a client disconnects and connects again later on with the same id, it is considered as a new client. It would be nice to be able to consider it as the same client (and so, its history of publications be recovered). This would also imply that one should be able to start following a disconnected client.