Yellit!

An application project for the course Computer Systems and Programming at la Sapienza held by Professor Giorgio Richelli

Alberto Arganese

Matr. 1916617

3/9/2020

Introduction

The goal is to write an application in C language where some users can interact exchanging messages in an argument based conversation open to everybody. The result was indeed similar to a forum website, where users can create conversations, ask questions, say their opinion waiting for other users response. The classic forum idea drove me for the construction of the application and my design choices.

A user can be registered and create a new topic or navigate through an existing one, viewing the threads relative to that topic and having the option to view the messages in a thread or write his own message. Record is to be kept for who created the topic or thread since only him can delete it (or an administrator).

User's submitted messages must be published by an admin, who can review and moderate them before making them public.

We were asked to use multi processes instead of threads and SYSV IPCs instead of other implementations.

My implementation presents a reduced number of .c files, since I realized the project using the "whiteboard" as the only container for our objects: the struct whiteboard will be directly linked with the other objects, without a fixed hierarchy. The main reason for this was that I could have exchanged updates with the database in a slightly easier way and I didn't want to push too much on object oriented programming.

Program structure

Four structures are available as header files. Whiteboard is the upmost layer and contains the lists of all topics, threads and messages available on the whiteboard. Moreover I found useful to have a counter for the current number of each of these objects.

```
typedef struct {
    Topic topicList[MAX_TOPICS];
    Thread threadList[MAX_THREADS];
    Message messageList[MAX_MESSAGES];
    int currentTopics;
    int currentThreads;
    int currentMessages;
} Whiteboard;
```

A topic has an id, a name and the id of its creator.

```
typedef struct {
  int idTopic;
  char topicName[64];
  int idOwner;
} Topic;
```

A thread has an id, the id of the topic on which is posted, a name, the id of its creator and the last message created.

```
typedef struct {
  int id;
  int idTopic;
  char threadName[64];
  int idOwner;
  Message lastMessage;
} Thread;
```

A message has an id, the status (0 if sent, 1 if published), the id of the thread in which is posted, the id of its creator and the content.

```
typedef struct{
   int idMessage;
   int status;
   int idThread;
   int idOwner;
   char messageText[1024];
} Message;
```

In the end we have the user, with id, username and password.

```
typedef struct{
   int id;
   char username[64];
   char password[64];
}User;
```

To preserve persistence of data, I created a file for each of these structures that will be our database.

To use these data in the whiteboard, I created 3 main functions:

populateTopics(Whiteboard* whiteboard)

populateThreads(Whiteboard* whiteboard)

populateMessages(Whiteboard* whiteboard)

After these 3 execute we will have all the info stored in our object whiteboard. The object whiteboard was created and stored in a shared memory, in order to obtaining the correct behaviour of the program and guarantee consistency between connected clients.

To avoid collisions, I created 3 SYSV semaphores: one for read, one for write, one to manage users.

The reason why I needed both a read and a write semaphore is that we can allow 2 clients to read in the same time from the shared memory, but not write together nor read while write. For this reason, I initialized the read semaphore value to 20 (max simultaneous reads) and the write one to 1, using semctl and the cmd SETVAL.

From now on, every read operation will just use a sem_op = -1 and decrease by 1 the semaphore value, allowing indeed 20 reads in the mean time, while for the write we will use the same sem_op = -1 but starting from val=1 we have exclusive access to the memory. The trick is that when we call a write operation, we do an additional step and launch a semop on the read semaphore too with sem_op = -20, allowing us to not access the memory if only 1 reader is present, and not permitting any reader to try to access the memory while we are writing.

The users semaphore is simpler instead, it works like the write semaphore but works independently, because I did not put the users from database to the shared memory.

To create the architecture (multi)client-server, in both Server.c and Client.c I created a socket and specified host and port. The server then, in an infinite loop waits for incoming connections and accepts them, creating a new process with fork that will communicate with the client, asking for an authentication.

Program execution

To run the program first Server.c has to be compiled and then run:

gcc Server.c -o Server

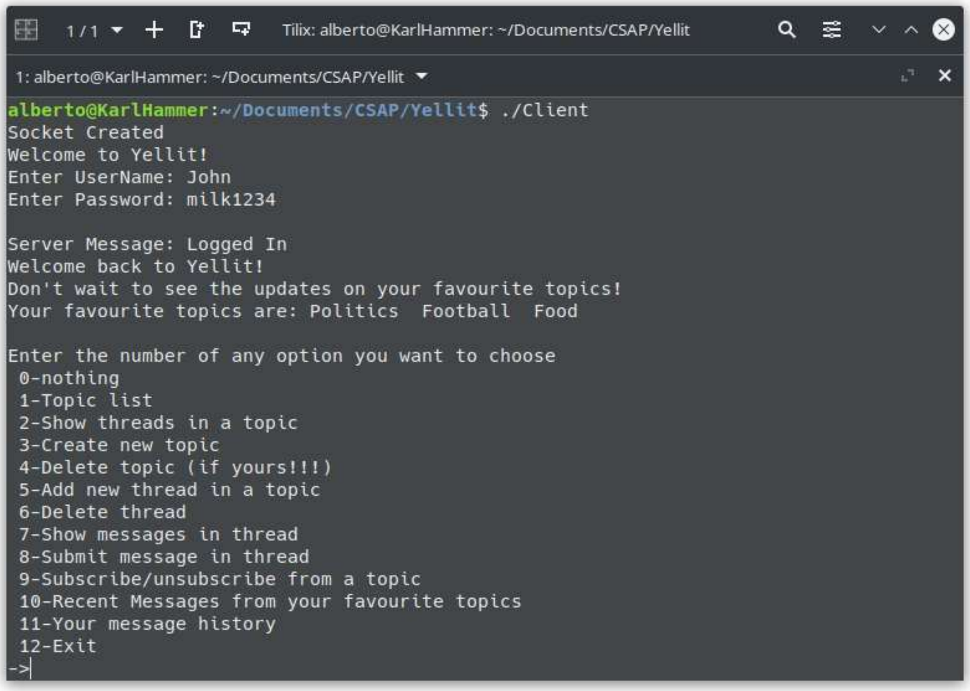
./Server

These commands will start the server side, waiting for connections from clients.

To run the client side:

gcc Client.c -o Client

./Client


1 / 1
Tilix: alberto@KarlHammer: ~/Documents/CSAP/Yellit
1: alberto@KarlHammer: ~/Documents/CSAP/Yellit

```
alberto@KarlHammer:~/Documents/CSAP/Yellit$ ./Client
Socket Created
Welcome to Yellit!
Enter UserName: John
Enter Password: milk1234

Server Message: Logged In
Welcome back to Yellit!
Don't wait to see the updates on your favourite topics!
Your favourite topics are: Politics  Football  Food

Enter the number of any option you want to choose
 0-nothing
 1-Topic list
 2-Show threads in a topic
 3-Create new topic
 4-Delete topic (if yours!!!)
 5-Add new thread in a topic
 6-Delete thread
 7-Show messages in thread
 8-Submit message in thread
 9-Subscribe/unsubscribe from a topic
 10-Recent Messages from your favourite topics
 11-Your message history
 12-Exit
->
```

After digiting username and password, the server will check if our credentials do match in the database with the function**: int authenticate(Whiteboard* whiteboard, char* username, int password).** It returns the id of our user (needed for other operations).

Normal user

The options represent:

1- Return to client the list of all topics.

2- Return to client the list of all threads in the topic the client will pass as input. If the user is not subscribed to that topic a message will warn him and he can't see those threads.

3- Create a new topic both in memory and in database, with name chosen by the user and id equal to last topic object id +1.

4- Client is requested to type the name of a topic he wants to delete, if the topic exists and the user is the original creator the operation succeeds and the topic is deleted from both the whiteboard in the shared memory and the database. To avoid hanging thread and messages, every thread and message under that topic will be deleted too.

5- User types the name of the topic he wants to insert the new thread in, then types the name of the new thread. The new thread is inserted in whiteboard and the database with creator user, id = last thread id +1.

6- User types the name of the thread he wants to delete and if he's the owner, operation succeeds and thread is deleted from whiteboard and database. To avoid hanging messages, every message under that thread will be deleted too.

7- User types the name of the thread and receives the list of all messages sent in that thread.

8- User replies to a thread writing a message, the server will insert it into whiteboard and database.

9- User types the name of the topic he wants to subscribe (or, if already subscribed to that topic, unsubscribe) and this information is updated in both whiteboard and database.

10- For every thread of a topic the user is subscribed, displays last message. This is not always true because I didn't implement in the database this function, so it's not persistent. I wanted it to be more like a "notification" that resets if system restarts, since it should be a fast information in a normal usage scenario when server is usually running and messages are appended to a thread with decent frequency.

11- Lists all the messages sent by the current user with status (published or sent).

Methods to accomplish above tasks

The first function calls the second, after both the operation is successful and a new topic is created in both db and whiteboard

*int addTopic(Whiteboard * whiteboard, char* topicName, int idOwner)*

int addTopicToDb(char topicName, int id, int idOwner)*

The first function calls the second, after both the operation is successful and a new thread is created in both db and whiteboard

*int addThread(Whiteboard * whiteboard, char* threadName, int idOwner, int idTopic)*

int addThreadToDb(char threadName, int idTopic, int id, int idOwner)*

The first function calls the second, after both the operation is successful and a new message is created in both db and whiteboard

*int addMessage(Whiteboard * whiteboard, char* messageText, int idOwner, int idThread)*

int addMessageToDb(char messageText, int idMessage, int idThread, int idOwner)*

The first one subscribe (or if already subscribed, it unsubscribes) the user from a topic. The second updates the database with that information.

int subscribeToTopic(int idUser, int idTopic)

int updateSubscriptions(int idUser)

Next functions are called in cascade starting from the first or the third, according if we're deleting the entire topic or a single thread.

int deleteTopic(Whiteboard whiteboard, char* topicName, int idUser)* *int deleteTopicFromDb(int idTopic)*

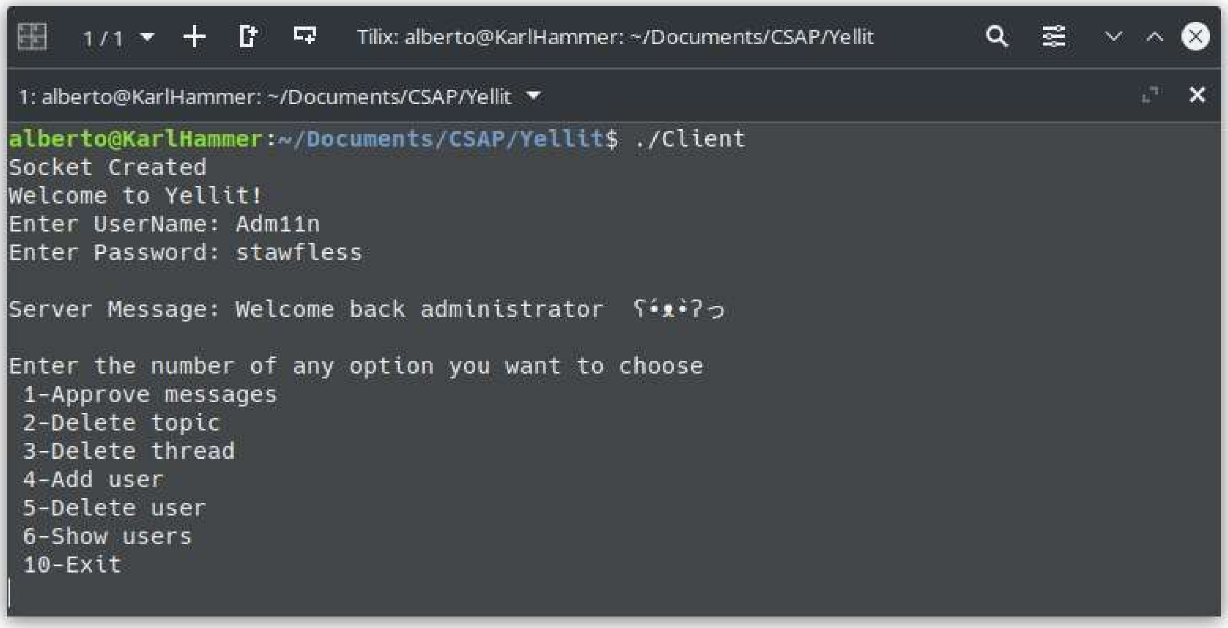
int deleteThread(Whiteboard whiteboard, char* threadName, int idUser)*

int deleteThreadFromDb(int idThread)

int deleteMessage(Whiteboard whiteboard, int idMessage)*

int deleteMessageFromDb(int idMessage)

Administrator


1 / 1
Tilix: alberto@KarlHammer: ~/Documents/CSAP/Yellit
1: alberto@KarlHammer: ~/Documents/CSAP/Yellit

```
alberto@KarlHammer:~/Documents/CSAP/Yellit$ ./Client
Socket Created
Welcome to Yellit!
Enter UserName: Adm11n
Enter Password: stawfless

Server Message: Welcome back administrator  ʕ•ᴥ•ʔっ

Enter the number of any option you want to choose
 1-Approve messages
 2-Delete topic
 3-Delete thread
 4-Add user
 5-Delete user
 6-Show users
 10-Exit
```

Options 2 and 3 are the same as the user 4 and 6, with the difference that the administrator can delete any topic or thread, even if he's not the owner. Other options are

1- Get a list of user's sent messages, the admin chooses which can be published (made visible to all users), the status of the message is updated in whiteboard and database.

4- Create a username by passing to server an id, username and password. Database updated

5- Delete a user by id. Database updated.

6- Get the list of all users from the database with their id, username and password.

Methods to accomplish above tasks

The admin see the messages sent by users and decides if they can be published. The method update the status of the message from 0 to 1 in the whiteboard message object and in database.

int publishMessage(Whiteboard whiteboard, int id)*

The admin gives id, username, password and the first method adds to “registered_users” the user credentials while the second creates a new row in “subscriptions_db”.

int addUser(int id, char username, char* password)*

int createRowSubs(int idUser)

Deletes user and his subscriptions from the 2 databases.

int deleteUser(int id)

int deleteRowSubs(int idUser)

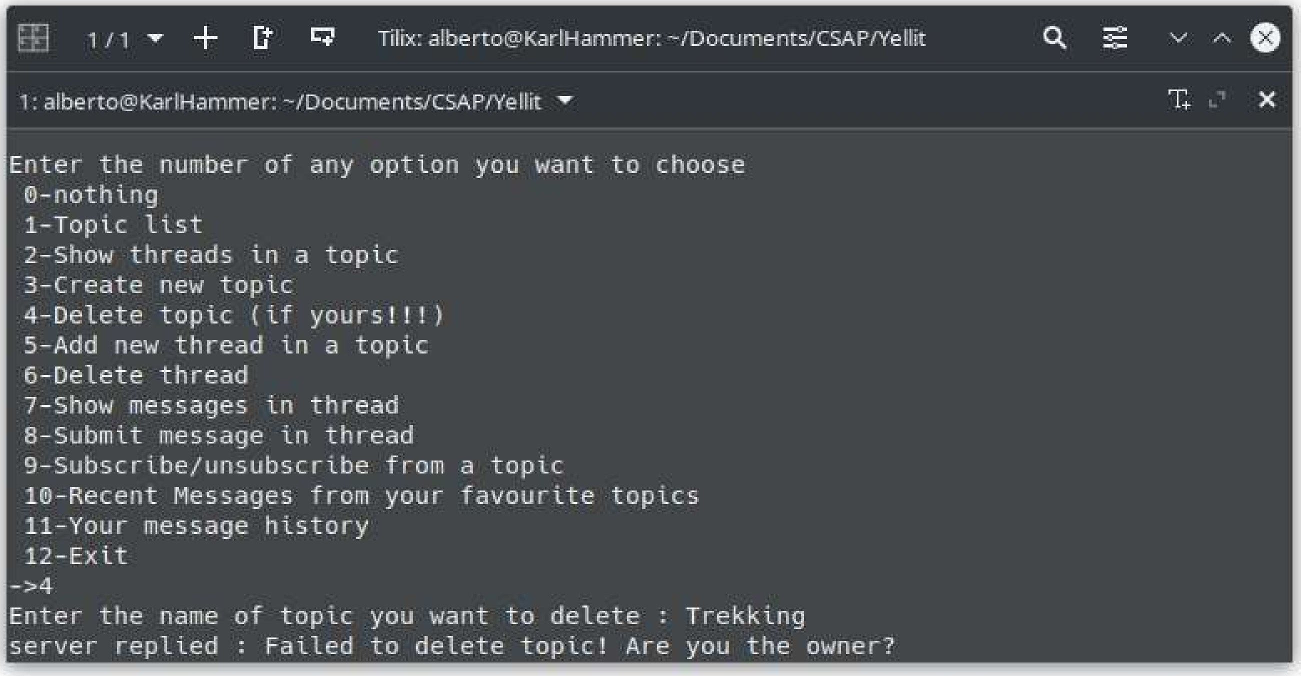
Show to admin the list of all users.

*char * showUsers()*

Input output control

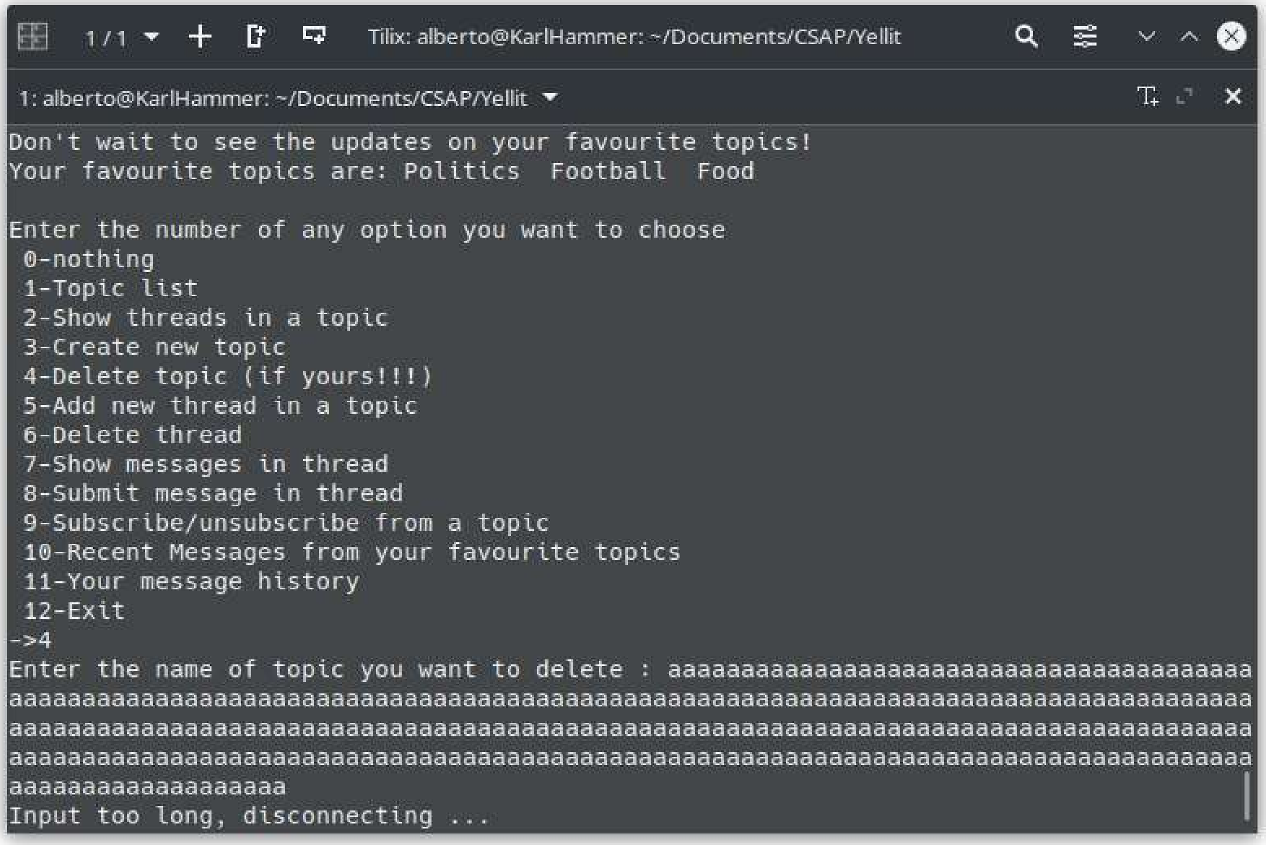
A lot of tests followed step by step the development of the application and helped me to understand more about my code and the possible frustration that an user could encounter if the server sent an error but without any information related to it.

In this example the user is not the creator of the topic, hence he cannot delete it. The server raise an error and warns the user.


1 / 1
Tilix: alberto@KarlHammer: ~/Documents/CSAP/Yellit
1: alberto@KarlHammer: ~/Documents/CSAP/Yellit

```
Enter the number of any option you want to choose
 0-nothing
 1-Topic list
 2-Show threads in a topic
 3-Create new topic
 4-Delete topic (if yours!!!)
 5-Add new thread in a topic
 6-Delete thread
 7-Show messages in thread
 8-Submit message in thread
 9-Subscribe/unsubscribe from a topic
 10-Recent Messages from your favourite topics
 11-Your message history
 12-Exit
->4
Enter the name of topic you want to delete : Trekking
server replied : Failed to delete topic! Are you the owner?
```

In the next example instead we have a very long input, and it is checked on client side, before even sending it to server. This is to avoid dangerous server behaviour since client is using fgets to fill client buffer, when the buffer dimension is exceeded the function overwrites it and the final “\n” is no more present, it can cause problems when reading the string on server side or client side.



```
Don't wait to see the updates on your favourite topics!
Your favourite topics are: Politics  Football  Food

Enter the number of any option you want to choose
 0-nothing
 1-Topic list
 2-Show threads in a topic
 3-Create new topic
 4-Delete topic (if yours!!!)
 5-Add new thread in a topic
 6-Delete thread
 7-Show messages in thread
 8-Submit message in thread
 9-Subscribe/unsubscribe from a topic
 10-Recent Messages from your favourite topics
 11-Your message history
 12-Exit
->4
Enter the name of topic you want to delete : aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaa
Input too long, disconnecting ...
```

Conclusions and difficulties

I'm not happy for one feature I would have implemented, a sort of live notification sending to all clients subscribed to a topic that a new messages has been sent in thread X. I managed to implement it with message queue and msgsnd, msgrcv and it worked, client received the notification on the terminal, but after that communication with server began to desynchronize by one message (server was in delay by one message). After a lot of try I had to give up and implement the option #10 in the user menu to show the latest message.

It has been a long run and in the end I'm satisfied with the final version of the project, because I never did a lot of C programming and the few things I knew had to be changed (for example the function gets unsecure and deprecated) and having in front of me a big project was intimidating at first but rewarding in the end. I never used semaphores nor shared memory before and I had to study a lot before even trying to copy some code from Stack Overflow.

I was inspired by the famous forum Reddit in doing this project (the name Yellit comes from that, and the fact that translated is “urlalo”, appropriate for a forum), so a possible addition to the application could be upvote and downvotes of messages and a profile panel to personalize, and of course live private messaging between users.