OS 2018

Ex4: WhatsApp-Like

Supervisor – Eytan Lifshitz

Due: 19.6.18

Part 1: Coding Assignment (90 pts)

Introduction

In this assignment you are required to deliver a WhatsApp-like service. The system is composed of one server and multiple clients. After a client connects to the service, it can send and receive messages from other clients.

Communication Protocol

In this exercise, you are required to create a simple communication protocol between the server and the client. This protocol should define how the server can know which command to execute (send, create_group, who, etc.), which client sends the command, etc.
You may find it useful to read https://en.wikipedia.org/wiki/Communications_protocol.

Server

The command line for running the server is: *whatsappServer portNum*
For example: *whatsappServer* 8875.

Notes:

- The server receives a port number and listens to this port. It waits for clients to connect to this port (using TCP connection) and serves their requests (more details below).
- When the server receives a request it needs to parse it and executes the request.
- In the real world when a server receives a new request from a client it must stay responsive for other clients. This can be achieved by using a different thread for the request processing. In this exercise you are not required to do so.
- The service is temporary – there is no need for the server to remember anything between different sessions, nor does it need to remember anything about unregistered clients (i.e. if client A is part of group X, then client A *exit*s and then *connect*s again, client A is no longer part of group X).
- At any time the user can enter EXIT in the stdin (keyboard) of the server. If the server process a command while the 'EXIT' is typed, it should finish processing this command (without starting to process another queued request), print "EXIT command is typed: server is shutting down" to server stdout and *exit(0)*.
  Note: typing 'EXIT' in the server stdin before any client connects the server is legal and should terminate the server.
- The server should manage the connection and stdin using **'select'**.

- During the server's activity, it should not have any memory leaks. It means that when a connection with a client is terminated, the server should release all the relevant resources.
- The maximal number of pending connections is 10 (the backlog parameter in the listen function).
- The server shouldn't crash upon receiving illegal requests.

Client

The command line for running the client is: whatsappClient *clientName serverAddress serverPort*
For example: whatsappClient Naama 127.0.0.1 8875.

Upon execution, the client will try to connect to the specified server. After a successful connection the client will print "Connected Successfully.\n" and wait for commands from its stdin.
If a client tries to connect and the given *clientName* is already in use, by a different client or group, the client should fail to connect to the server. In this case the client will print "Client name is already in use.\n" and *exit(1)*.
If a client fails to connect for any other reason it should print "Failed to connect the server" and *exit(1)*.
After a successful connection of client *clientName* the server should print "*clientName* connected.\n".

The client will support all the commands specified below. A command is defined to be one line, i.e. all the text typed until an ENTER key is pressed is considered as a single command.

<u>Notes</u>:
- Commands, clients and group names should all be case sensitive.
- A group name can only include letters and digits.
- A client name can only include letters and digits.
- All the connections between the client and server are TCP based.
- The client should check user input before sending it to the server (i.e. valid commands, expected and valid arguments, etc.). The client should send only valid requests.
- The received *serverAddress* is an IP address (and not a DNS address).
- In case the server exit before the client, the client should *exit(1)* without crashing.

Client Commands:

The client should allow the following commands:

Command	Arguments	Command description	Server output	Sender client output
create_group	<group_name> <list_of_client_names>	Sends request to create a new group named "group_name" with <list_of_client_names> as group members. "group_name" is unique (i.e. no other **group or client** with this name is allowed) and includes only letters and digits. <list_of_client_names> is separated by comma without any spaces. For example:	<u>Upon success</u>: <sender_client_name> : Group "<group_name>" was created successfully.\n <u>Upon error:</u> <sender_client_name> : ERROR: failed to create group "<group_name>".\n	<u>Upon success</u>: Group "<group_name>" was created successfully.\n <u>Upon error:</u> ERROR: failed to create group "<group_name>".\n

		● create_group osStaff David,Eshed,Eytan,Yair Notes: ● It's an error to create a group without members or with invalid members. ● The client who sends the request should be part of the group (even if it isn't part of the received list). ● A group must contain at least two members (including the client who creates the group). ● If client name appears more than one time in the received list, consider it as one instance. For example: calling "create_group osStaff Eshed,Eshed,Eytan" is the same as "create_group osStaff Eshed,Eytan".		
send	<name> <message>	If *name* is a client name it sends *<sender_client_name>: <message>* only to the specified client. If *name* is a group name it sends *<sender_client_name>: <message>* to all group members (except the sender client). Notes: ● Only a group member can send a message to the group. ● The received *name* must be different than the sender name (i.e. it's invalid to send message only to yourself).	<u>Upon success</u>: <sender_client_name> : "<message>" was sent successfully to <name>.\n <u>Upon error:</u> <sender_client_name> : ERROR: failed to send "<message>" to <name>.\n	<u>Upon success</u>: Sent successfully. \n <u>Upon error</u>: ERROR: failed to send.\n
who		Sends a request (to the server) to receive a list of currently connected client names (alphabetically order), separated by comma without spaces.	<sender_client_name> : Requests the currently connected client names.\n	<u>Upon success</u>: <ret_client_names_separated_by_comma_without_spaces>.\n

				Upon error: ERROR: failed to receive list of connected clients.\n
exit		Unregisters the client from the server and removes it from all groups. After the server unregistered the client, the client should print “Unregistered successfully” and then *exit(0)*.	<sender_client_name> : Unregistered successfully.\n	Unregistered successfully.\n

Notes:

- You can assume that the maximal client name and group name are 30 characters each, the maximal message length is 256 characters and the maximal group size is 50 clients (no need to check it).
- You can assume that client names and group names have no spaces in them (no need to check it).
- You can assume that every client calls the ‘exit’ command before closing.
- Every message should be printed in a new line.

An example of the commands above is:

Server input	Server output	client1 input	client1 output	client2 input	client2 output
whatsappServer 8875					
		whatsappClient client1 127.0.0.1 8875			
	client1 connected.		Connected successfully.		
				whatsappClient client2 127.0.0.1 8875	
	client2 connected.				Connected successfully.
		create_group firstG client2			
	client1: Group “firstG” was created successfully.		Group “firstG” was created successfully.		
				create_group client1 client1 [Note: client1 is the name of the group – illegal request]	
					ERROR: failed to create group “client1”. [Note: nothing is printed in the server as the client can catch the error before sending the command]
				create_group secG client2 [Note: there are less than 2 members – illegal request]	
					ERROR: failed to create group “secG”. [Note: nothing is printed in the server as the client can catch the

					error before sending the command]
		send client2 how are you?			
	client1: “how are you?” was sent successfully to client2.		Sent successfully.		client1: how are you?
		send firstG hi again			
	client1: “hi again” was sent successfully to firstG.		Sent successfully.		client1: hi again
				who	
	client2: Requests the currently connected client names.				client1,client2
		exit			
	client1: Unregistered successfully.		Unregistered successfully.		

Error Handling:

- In case of server usage error you should print the following message (to the server std output): “Usage: *whatsappServer portNum\n*“
- In case of client usage error you should print the following message (to the client std output): “Usage: whatsappClient *clientName serverAddress serverPort\n”*
- You are required to check user inputs in the client side. If the received command doesn’t exist, print “ERROR: Invalid input.\n”. Otherwise print the relevant error message from the table. In both cases <u>don’t send</u> it to server.
- In case of a system call error in the server side, print “ERROR: <system call function name> <errno>.\n” (for example: “ERROR: accept 22.\n”)
- In case of a system call error (e.g., if the program failed to open socket) in the client side, the program will do the following:
    - Print “ERROR: <system call function name> <errno>.\n” (for example: “ERROR: socket 24.\n”)
    - Then exit the client program with *exit(1)*.
- You are **not** required to deallocate all the resources in case of a system call error.

Background reading and Resources

Read the following man-pages for a complete explanation of relevant system calls and functions: socket, bind, connect, listen, accept, select, send, recv, close, inet_addr, htons, ntohs, gethostbyname, setsockopt.

In order to make your life easier and help you focus on sockets rather than string processing and printing to screen, we wrote for you the library whatsappio.h which can save much of the hassle of the unimportant issues of this exercise. You are encouraged (although not have to) use it.

Guidelines

Note that the protocol between the clients and the server is up to you. In other words, you decide exactly what format to use and when passing information from one to the other.
Don’t forget to check the return value of all system calls.

Part 2: Theoretical Questions (10 pts)

The following questions are here to help you understand the material. We're not trying to trick or fail you, so answer straight forward.

1. You are required to add one more command: "leave_group <group name>" – the client will be removed from the specified group. If the group becomes empty it should be removed (i.e. it will be valid to use <group name> again).
    a. Which changes are required in the client side in order to support the new command?
    b. Which changes are required in the server side in order to support the new command?
2. Why do we prefer TCP over UDP in this exercise?
3. Give two examples for applications that use UDP. Why is UDP preferred for them?
4. In this exercise, in case of a server crash all the data is lost (we say that the server is a single point of failure). Describe high level design for a system that is able to recover from failures (we allow some of the data to be lost).

Submission

Submit tar file named ex4.tar containing:

1. README file built according to the course guidelines. Remember to add your answers to the README file.
2. Source code including *whatsappServer*.cpp and *whatsapp*Client.cpp. If you choose to use whatsappio.h or whatsappio.cpp submit them as well.
3. A Makefile which compiles the executables. That means that a simple "make" command creates two executables: *whatsappServer* and *whatsapp*Client. More requirements of the Makefile appear in the course guidelines.

Make sure that the tar file can be extracted and that the extracted files do compile.

Late Submission Policy

Submission time	**21.6, 23:55**	**24.6, 23:55**	**25.6, 23:55**	**26.6, 23:55**	**27.6, 23:50**
Penalty	0	3	10	25	40