OS 2016

Ex5: Events Management

Supervisor – Tal Orenstein

Due to: 16.6.16

Part 1: Coding Assignment (90 pts)

Introduction

In this assignment you are required to deliver an events management system. The system is composed of one server and multiple clients. After a client registers to the service, it can ask to receive details about new events created by other registered clients, RSVP for an event, asks to get all clients that RSVP'ed for specific event and more.

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 (register, create a new event, 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: *emServer portNum*
For example: *emServer* 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.
- When the server receives a new command from a client it is required to execute it on a new thread.
- The server should also maintain a log file named emServer.log, which includes all the commands it received. See commands table below and 'Server Log' section for more details.
- The service is temporary – there is no need for the server to remember anything between different sessions, nor doesn't need to remember anything about unregistered clients (i.e. if client A RSVP'd for an event X, then client A *unregistered* and then *registered* again, client A no longer RSVP'd to event X). Below you can find more details about *unregister* command.
- 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 start processing other queued requests), print "EXIT command is typed: server is shutdown" to server log, close this log and *exit(0)*.
  Note: typing 'EXIT' in the server stdin before any client connects the server is legal and should terminate the server (you can set a timeout of 2 seconds).
- During the server's activity, it should not have any memory leaks. This 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 event ids should be sequential (start from 1).

Client

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

Upon execution the client will wait for commands from stdin (keyboard). 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 a single command.
In addition, the client should also maintain a log. For more details see 'Client Log' section below.
In case that the client fails to connect the server, print the error to log (see 'Error Handling') and *exit(1)*.

Notes:

- Commands and client name reference should all be case insensitive.
- It is considered as an error to send commands (other than 'REGISTER') to the server before performing 'REGISTER' operation successfully.
- It is considered as an error to send commands (other than 'REGISTER') to the server, after the server performs 'UNREGISTER' operations successfully.
- For more details about these commands, see below.
- All the connections between the client and server are TCP based.
- It's OK (and recommended) not to send invalid commands to the server (none existing commands, missing arguments, etc.). In this case, you should print an appropriate error message to the client log (see 'Client Log' section for more details).
- The received *serverAddress* is an IP address (and not a DNS address).

Client Commands:

The client should allow the following commands:

Function	Command	Arguments	Command description	Server Log format	Client Log format
Register	REGISTER		Register a new client on the server. The client passes its name to the server. If the client name already exists in the server (case insensitively), the client should print "ERROR: client <client name> was already registered." to its log and then *exit(1)*.	Upon success: <Client name>\t was registered successfully.\n If the client name is already registered: ERROR: <Client name>\t is already exists.\n	Upon success: Client <client name> was registered successfully.\n If the client name is already registered: ERROR: the client <client name> was already registered.\n
Create a new event	CREATE	eventTitle\s eventDate\s eventDescription	Sends the received *event's title, date* and *description* to the server (you may want to send more data,	Upon success: <Client name>\t event id <event	Upon event id arrival: Event id

			according to your protocol). The server returns an event id to the sender client. Note: Using whitespace as part of the title and date is prohibited. An example of this command: CREATE my_event_title event_date event description	id> was assigned to the event with title <event title>.\n	<eventId> was created successfully.\n <u>In case of an error</u>: ERROR: failed to create the event: <error description>.\n
Get top 5 newest events.	GET_TOP_5		Sends request to receive a list of the top 5 newest events (id, title, date and description). The list should be sorted from the newest to the oldest (the events with the top 5 ids).	<Client name>\t requests the top 5 newest events.\n	<u>Upon success</u>: Top 5 newest events are:\n<list of events>.\n <u>Note</u>: see comment regarding <list of events> below. <u>In case of an error</u>: ERROR: failed to receive top 5 newest events: <error description>.\n
Send RSVP	SEND_RSVP	eventId	Sends client RSVP for the event with id *eventId* to the server (i.e., client X is going to event id Y). If the client didn't send an RSVP message already, the server needs to add the client to the RSVP's list of this event id. An example of this command: SEND_RSVP 4	<Client name>\t is RSVP to event with id <event id>.\n	<u>Upon success</u>: RSVP to event id <event Id> was received successfully.\n <u>If the client already sent RSVP message for this event</u>: RSVP to event id <event Id> was already sent.\n <u>In case of an error</u>: ERROR: failed to send RSVP to event id <event Id>: <error description>.\n
Get RSVP's List	GET_RSVPS_ LIST	eventId	Sends request to receive a list of client names that RSVP'd to event id *eventId*. Upon receiving a list (may be an empty list), the client is required to sort it alphabetically and prints an appropriate message to its log. Note: The client names in the sorted list should be separated by comma without whitespaces. For example: "The RSVP's list for event id 1 is: David,Netanel,Lirane,Tal".	<Client name>\t requests the RSVP's list for event with id <event id>.\n	<u>Upon receiving a list</u> (may be an empty list): The RSVP's list for event id <event Id> is: <the sorted list>.\n

			An example of this command: GET_RSVPS_LIST 4		
Unregister	UNREGISTER		Unregisters the client from the server and removes it from all RSVP lists. If the server unregistered the client successfully, prints the message shown in the 'client log format' column. Then *exit(0)*. On failure, prints an error message to client log and **don't** exit the program.	<Client name>\t was unregistered successfully.\n	Upon success: Client <client name> was unregistered successfully.\n

Note:
You can assume that the maximal event title and date lengths are 30 characters each and the maximal event description length is 256 characters (check it on the client side and print an error message in the case of a violation of this assumption).

Server Log

The server should maintain a log file named emServer.log, which includes all the commands it received.
Each line in the log should start with the time in HH:MM:SS format, followed by '\t'.
Any error that happens in the server side should be printed to the server log. See 'Error Handling' section for the specific format.

Notes:

- The client should not send illegal commands/commands with missing or invalid arguments to the server (they can be identified on the client).
- Every message should be printed in a new line.
- All the server outputs should be printed to this log.

Here you can find an example for possible server log.

Client Log

The client should also maintain a log file named <client_name>_HHMMSS.log (for example: if the client name is "tal" and the current time is 15:32:12, the log name is "tal_153212.log").
All the prints of the client should be written to this log.

Each line in the log should start with the time in HH:MM:SS format, followed by '\t'.

Notes:

- Every message should be printed in a new line.
- Regarding <list of events> (relevant for 'GET_TOP_5' command): each event in the list should be printed in the following format:
  <event id>\t<event title>\t<event date>\t<event description>.\n
- All the client outputs should be printed to this log.

Here you can find an example for possible client log.

Error Handling:

- In case of server usage error you should print the following message (to the server std output): "Usage: *emServer portNum*"
- In case of client usage error you should print the following message (to the client std output): "Usage: emClient *clientName serverAddress serverPort"*
- You are required to write an error message to the client log for every error in the client side:
    - Illegal command: "ERROR: illegal command.\n"
    - Missing arguments: "ERROR: missing arguments in command <command>.\n"
    - Invalid argument: "ERROR: invalid argument <argument> in command <command>.\n"
    - Sending command other than 'REGISTER' while the client is not registered yet: "ERROR: first command must be REGISTER.\n"
- You are required to write an error message to the server log for every error in the server side.
- You are required to check user inputs and write to client log an appropriate error message in case of invalid usage (for example: missing arguments, invalid argument, etc.).
- **[5.6]** Every error message in the server log (except system call error) should be in the following format:
  HH:MM:SS \tERROR\t<function name>\t<error description>.\n
- **[5.6]** In case of a system call error in the server side, don't include the client name:
  HH:MM:SS \tERROR\t<system call function name. For example: accept>\t<errno>.\n
- **[5.6]** Every error message in the client log should be in the following format:
  HH:MM:SS \tERROR\t<function name>\t<error description>.\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:
    - Prints an error message.
    - 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, strftime, setsockopt.

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 when passing information from one to the other. Basic guidelines are as follows:

- Any component that recognizes an error condition (other than those detailed above) should print an informative error message to its log. This should start with "Error: ". If a server error also concerns the client (e.g. when a client command fails), the client should print it as well.
- 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 don't try to trick or fail you, so answer straight forward.

1. You are required to add one more command: "Create and Distribute" – after creating a new event, the server is required to distribute the event to all registered clients.
    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 UDP is 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 ex5.tar containing:

1. README file built according to the course guidelines. Remember to add your answers to the README file.
2. Source code including *emServer*.c / *emServer*.cpp and emClient.c / emClient.cpp.
3. A Makefile which compiles the executables. That means that a simple "make" command creates two executables: *emServer* and emClient. 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	**16.6, 23:55**	**17.6, 11:50**	**18.6, 23:55**	**19.6, 23:55**	**20.6, 23:50**	**Later**
Penalty	0	3	10	25	40	**Course failure**