CS 421 – Computer Networks
Programming Assignment I
SecuritySystem

Due: 15 March 2018

In this programming assignment, you are asked to implement the client side of a simple server-based security system. Your program must be a console application (no graphical user interface (GUI) is required) and should be named as *SecuritySystemClient* (i.e., the name of the class that includes the main method should be *SecuritySystemClient*). The server code is provided with the assignment.

Security Server Protocol

SecuritySystemServer uses a custom application level protocol which is built on top of TCP to communicate with the client. The format of this protocol is illustrated below:

Figure 1: Illustration of the format of messages in the custom protocol that the security system uses.

The first byte of the message contains a number which indicates the type of the message. Possible message types are as follows:

Table 1: Message types and their representative values.

Message Type	Value
AUTHORIZE	0
KEEPALIVE	1
OK	2
INVALID	3
EMERGENCY	4
ALARM	5
DISCARD	6
EXIT	7

E.g., according to Table 1 above, to send an ALARM message, the first byte of the message to be sent should be set to 5 (0000 0101 in bits).

The next 2 bytes of the message indicates the length of the data (in bytes) that is sent with the message. E.g, if server sends a 12 kilobytes photo that is captured by the security camera to the client, the Data Length field should be set to 12 kilobytes (12 288 in decimal, 0011 0000 0000 0000 in bits). Note that, if there is no data in the message, this field should be set to 0.

Finally, the rest of the message after the first 3 bytes includes the data that is sent with the message (if there is any). This field has a variable length.

Client Program

Server and client communicate using the Security Server Protocol explained above, over TCP as the transport layer protocol. You will use the Socket API of the JDK to implement the application level protocol that is described in the previous part in order to establish communication between the server and the client.

The server program that is provided with the assignment runs with the following command in the console:

java SecuritySystemServer <port_number>

where *<port_number>* is the port number to which the socket is bound. After executing this command, the server program runs and starts to listen for incoming TCP segments from the specified port.

The client program that you are asked to implement should run with the following command:

java SecuritySystemClient <port_number> <username> <password>

where *<port_number>* is the port number to which the server socket is bound, <username> is the username and <password> is the password required for authentication. Note that since you will be running both the server and the client on the same computer (for the sake of simplicity), you can assume that server and client share the same IP address.

After the server starts running, you will execute the client program with the command described above. As soon as the client program starts running, it should send an AUTHORIZE message to the server with its data field filled with the following information

<username>:<password>

which should be encoded in **US-ASCII** standard. Note that the username and the password are

hardcoded within the server program as:

username: *bilkent*

password: *cs421*

Hence, the data field of the AUTHORIZE message should be *bilkent:cs421* (encoded in US-ASCII). Although the authentication information is fixed for this assignment, DO NOT hardcode the username and the password inside your client program; these values should be taken from the user as command line arguments.

Once the client sends the authentication information to the server, it should wait for the response of the server. There are two possible responses:

- **OK**
- **INVALID**

These response messages do not contain any data. If correct authentication information is entered, then the server should send an OK message back to the client.

If the client receives an OK message, then it starts to listen for possible messages that will arrive from the server. These messages are explained below:

- **KEEPALIVE**: This message includes no data. It is sent from the server to the client at regular time intervals (e.g., once every 3 seconds) in order to make sure that the connection between the server and the client persists.
- **EMERGENCY**: This message indicates that there is a possibility of intrusion; so, the server sends a snapshot taken from its security camera along with the message. The server sends this message when some random amount of time has elapsed after the program starts running.
- **EXIT**: This message indicates that the server is shutdown; so, the client program can terminate. The server is programmed to simulate a shutdown after sending 2 EMERGENCY messages.

Your program should take different actions depending on the type of message received from the server. These actions are explained below:

1. If the client receives **KEEPALIVE**: The client should send another KEEPALIVE message back to the server so that the server can be certain that it is still connected to the client. After sending the KEEPALIVE message back to the server, the client should check whether the server responds with an **OK** message. The client should continue listening to incoming

messages from the server after making sure that the server sends OK. Note that server might also send an **INVALID** message as the response, which means there is something wrong with the client's message. However, you are not asked to take any action in case of an INVALID response, since you are asked to write a client program which works correctly and does not receive any INVALID messages.

2. If the client receives **EMERGENCY**: Since EMERGENCY message also includes data, the client should receive all the data inside this EMERGENCY message from the server. E.g., the server will send a photo (as a text file encoded in **US-ASCII**) to the client in the following format:

139 128 108 97 88 86 105 118\n
107 116 109 111 104 94 96 88 \n
89 119 131 142 168 170 179 201\n
221 183 134 130 133 150 157 160\n
162 164 167 170 173 176 179 182\n
185 188 192 196 200 204 208 211\n
212 214 215 214 215 215 255 255\n

where each number represents a pixel value. Each row is separated by a newline (\n) character and each column is separated by a white space character. The client program should receive the image file from the server and save it in a text file called "**snapshot_n.txt**", where *n* is an integer starting from 1 and incremented each time a new EMERGENCY message received (e.g. snapshot_1.txt, snapshot_2.txt, …, snapshot_6.txt…).

After the EMERGENCY message and its content are received, the server waits for the response of the client. There are two possible messages that the client can send at this point:

- Send an **ALARM** message (meaning, there is indeed an emergency)
- Send a **DISCARD** message (meaning, there is no need to take any action)

Your program should ask for user input to take the necessary action. If the user enters **1**, it should send an **ALARM** message. If the user enters **2**, it should send a **DISCARD** message. Note that these messages do not contain any data. An example run for the EMERGENCY case is given below:

>> *Emergency message received. Enter 1 to ring the alarm, enter 2 to discard:*

>> *1*

>> *Alarm message sent.*

or

>> *Emergency message received. Enter 1 to ring the alarm, enter 2 to discard:*

>> *2*

>> *Discard message sent.*

After sending ALARM or DISCARD message, your program again should check the server response. If it is OK, it should continue to listen to incoming messages from the server. Similarly, you might receive an INVALID message in this scenario as well, which means your client does not send correct messages to the server. Make sure to debug your program accordingly, so that you receive OK as the responses.

3. If the client receives **EXIT:** The program should terminate.

A state diagram that summarizes the client behavior is given below:

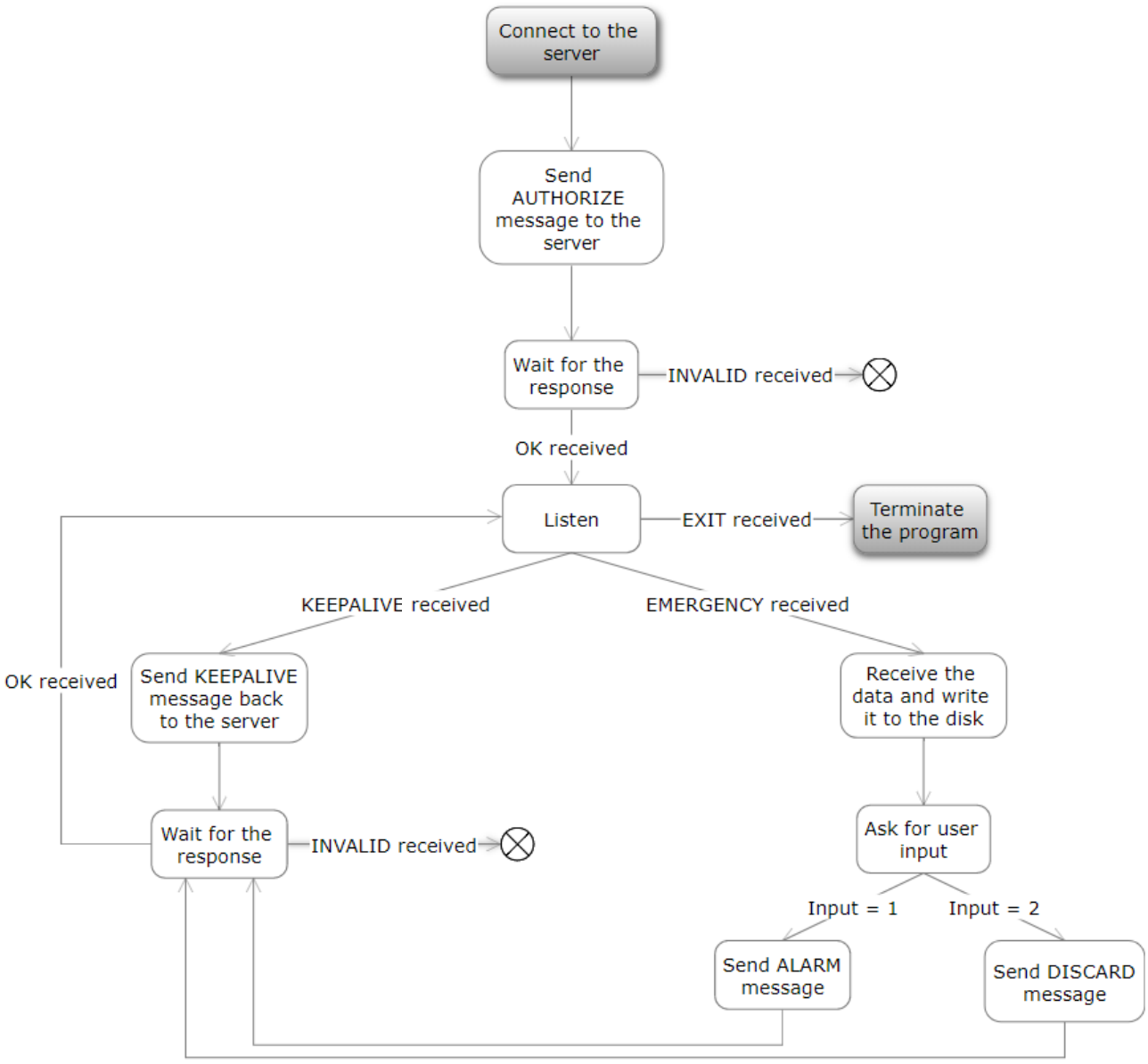

Figure 2: State diagram for the client.

Note that the client **always** checks the response of the server after it sends a message to the server. In the state diagram above, INVALID server responses are shown as crosses (X), since you should not receive any INVALID messages if your client program sends correct messages to the server. Server prints an error message on the console just before sending an INVALID message and terminates the program if it receives an incorrect message.

Submission Rules

You can do this assignment either individually or together with another student from the **same** course section.

You need to apply all of the following rules in your submission. **You will lose points if you do not obey the submission rules below or your program does not run as described in the assignment above.**

- The assignment should be submitted as an e-mail attachment sent to bulut.aygunes[at]bilkent.edu.tr. Any other methods (Disk/CD/DVD) of submission will not be accepted.
- The subject of the e-mail should start with [CS421_PA1], and include your name and student ID. For example, the subject line should be

  [CS421_PA1]AliVelioglu20111222

  if your name and ID are Ali Velioglu and 20111222. If you are submitting an assignment done by two students, the subject line should include the names and IDs of both group members. The subject of the e-mail should be

  [CS421_PA1]AliVelioglu20111222AyseFatmaoglu20255666

  if group members are Ali Velioglu and Ayse Fatmaoglu with IDs 20111222 and 20255666, respectively.
- All the files must be submitted in a zip file whose name is the same as the subject line **except** the **[CS421_PA1]** part. The file must be a .zip file, not a .rar file or any other compressed file.
- All of the files must be in the **root** of the zip file; directory structures are not allowed. Please note that this also disallows organizing your code into Java packages. The archive should not contain:

  – Any class files or other executables,

  – Any third-party library archives (i.e., jar files),

  – Any text files,

  – Project files used by IDEs (e.g., JCreator, JBuilder, SunOne, Eclipse, Idea or NetBeans, etc.). You may, and are encouraged to, use these programs while developing, but the end result must be a clean, IDE-independent program.

- The standard rules for plagiarism and academic honesty apply. For a discussion of academic integrity, please read this document. Refer to 'YÖK Student Disciplinary Rules and Regulations', items 7.e and 7.f, and 'Bilkent University Undergraduate Education Regulations' item 4.9 for possible consequences of plagiarism and cheating.