UNIVERSITY OF TARTU, INSTITUTE OF COMPUTER SCIENCE

Task 2: TCP Communication

Hadachi&Lind

October 12, 2017

Must Read:
The task 2 should be done individually! You can submit your solution for task using course page:
https://courses.cs.ut.ee/2017/ds/fall/Main/Seminars
Regarding the submitted code:

- The solution should include your source code created and Readme.txt file explaining how to run it.
- NB! The solution has to be programmed in Python 2.7.x.
- NB! You are free to reuse the example code from the seminars.
- NB! You are free to write **your own implementation** from scratch (as long as you do it in Python).
  - Try to write your own protocol on top of TCP using Socket API of Python 2.7. In other words, try to avoid using libraries that will handle TCP or wrap the Socket API.
  - Regarding the usage of Threads, OOP, GUI etc. - there are no strict limitations

1 IMPLEMENTING SIMPLE FILE TRANSFER PROTOCOL

During the last seminar we have discussed TCP and its advantages. In this task, we will put it into use by implementing step by step our own file transfer protocol on top of TCP. In order to achieve our goal we will introduce three main steps. And they are as follows:

1.1 STEP 1

We will create a simple dump-server, where the client can only upload files. The sequence of the process can be resumed as follows: client connects, uploads the file, and disconnects. They will be no control on the server-side. This means no maximal file size limits and no check if the server runs out of disk space.

In order to give a clear idea about the process sequence in details, here is how it should be:

Client-side:

1. User selects the file to upload
2. Client connects to server
3. Client opens the file for reading
4. Server reserves the file using randomly generated name, and prepares to write into file
5. Client uploads the data from the file, server stores the data into file
6. Client acknowledges end of upload using socket shutdown routine, server discovers end of transmission (empty receive buffer) and closes the file.
7. Client closes the file
8. Client disconnects (closes the socket)
9. Client application terminates

Server-side:

Sequential: clients are processed one after another[1]

1. User selects the directory where the dump should stored (dump-directory)
2. Server creates listener-socket and falls into endless serving loop:

   a) accept client's connection

   b) generate random name for client's file, and open the file for writing in the dump-directory

   c) receive and store the blocks of data and store them into file till the receive buffer is empty (which means did issue shutdown)

   d) close the client's file

   e) close the client's socket

In case user wishes to interrupt the endless serving loop and terminate the server application, we assume SIGINT or SIGTERM will be sent. In this case, before terminating server has to:

[1] We did not yet introduce Threads and synchronization. But if you feel confident to use them, then you are free to program the Thread-based client processing (if you do so do not forget to take care of the Thread safety as well).

- Shutdown the client socket and close it
- Close the listener-socket

In case of communication errors, both client and server should take into account the possible socket exceptions, for example in case of “Broken pipe error”:

- Client should:
  - report the communication error to User
  - close socket
  - close the file
  - terminate
- Server should:
  - log the communication error
  - close client socket
  - close the file
  - remove unfinished file
  - proceed with serving next client

At this point it should be clear that:

- There is no need for request headers, since there is only one kind of them: upload request.
- There is no need for request arguments, since we do not transfer anything but the raw data from the file.
- There is no need for additional markers denoting end of transmission, since we rely on socket shutdown routine.
- There is no need for responses (neither negative nor positive), since we rely on exceptions that can be thrown. Thus, we assume the transfer was successfully done if no errors were thrown.

1.2 Step 2

In the step 2, we will improve the previous step 1.1 by adding a simple protocol in order to preserve the file names. the protocol is as follows:

1. Before sending the data the client should first send the file name and wait for acknowledgment from the server
2. Server should check if no file with such name exists already in the dump-directory and

a) if there is no file with such name in the dump directory, the file with such name is created and opened for writing. Next, the server sends acknowledgment to a client, and client may start sending the data.

b) if there is already a file with such name server should acknowledge the client, that the file with such name can not be uploaded because it already exists. Client should show the warning message to the User and disconnect from the server.

At this level, we still have only one request, but we need to introduce parameter (the file name). In addition, we need two responses status that the server may reply with to the client after checking the file name existence:

- upload can start
- upload cannot start, file with such name already exists

NB: Be aware that the server first reads the message containing the file name to upload. Server must understand where to stop reading (fixed size message containing file name or use terminator character at the end of the file name string). Once the server discover the terminator character, it should stop reading the file name string from the socket, do the file name check, and then answer back to the client about check status. In case the check status is positive, then the server should be ready to receive the file.

1.3 STEP 3

This step is about improving the step 1.2 by making the client checking on the server, if it is ready to store this amount of data (file size) before starting the upload.

1. Before sending the data, the client should first send the file name and the size of the file to be uploaded. Then, the client should wait for acknowledgment from the server.
2. Server should check the file name as was specified in step 1.2
3. Server should check the size of the file provided by client (server should check the free space currently available for the dump-directory).

   a) if there is enough space for storing the file, then the server sends acknowledgment to a client, and client may start sending the data.

   b) if there is not enough space for storing the file, server should acknowledge the client that there is not enough space and the client should show the warning message to the User and disconnect from the server.

At this point, we still have only one request, but we need to extend parameters (we need to send both: file name and size). Also we need another negative response in case there is not enough space on server:

- upload cannot start, there is not enough space on server

1.4 Step 4

In the step 4, we will improve the step 1.3 by extending protocol and introducing the feature of letting the user requesting the list of all the files currently uploaded. Therefore, the client application should give user a choice either to upload the file or to list the files that are already stored on the server. For this reason, the server has to be able to distinguish between an “uploading” request or “listing files” request based on their header.

[you may use two control codes (0 or 1, for example); one is for “uploading” request and the second one for “listing files” request. (0 refers to “uploading” / 1 refers to “listing files”)]

- In case User decides to upload the file
  - Client sends the corresponding request
    - The arguments (file name and size) have to be sent with the control code for uploading
  - Client application continues performing as specified in steps 1.1, 1.2 and 1.3.
- In case User decides to list files
  - Client sends corresponding request
    - No additional arguments (only control code for listing)
  - Server issues directory listing on dump-directory, collecting the list of file names in this directory
  - Server sends the response with the list of file names
    - Here we may avoid special response codes, we may just send the comma-separated (or semicolon-separated) list of files in response, however of the situation where there no files (we cannot just send empty string).

1.5 Step 5

Finally, we add to the existing requests from step 4 (“uploading” and “listing”) a downloading request. The downloading feature allows the user to download the file by name provided by the user.