Networks & Operating Systems Essentials 2 (NOSE 2)

Assessed Exercise 1: Networking

The aim of this exercise is to have students use the knowledge they've acquired in this first part of the course, in the context of building a networked application. You will be using the Python socket library to create a simple, yet powerful, file service application. Your application will consist of two Python scripts: (a) a "server" that receives and serves client requests for files stored in a local directory, and (b) a "client" that allows the user to upload/download files from the server, as well as list the files currently stored on the server side.

Python Sockets

An Internet socket is an abstract representation for the local endpoint of a network connection. Berkeley sockets is an API for Internet sockets, coined after the first implementation of sockets appeared in 4.2BSD (circa 1983). Over time, Berkeley sockets evolved to what is now known as POSIX sockets – an IEEE standard defined with the aim of maintaining compatibility and providing interoperability between operating systems. POSIX sockets can be used to communicate using a number of protocols, including (but not limited to) TCP, UDP, ICMP, SCTP and others.

The basic workflow for setting up a TCP connection to a remote host using POSIX sockets was outlined in the course lectures and in the 3rd lab; please consult the lecture slides/recordings and lab handouts. For this assessed exercise you can use all methods provided by the basic Python socket library (`Lib/socket.py`). You are **not** allowed to use any other networking libraries that act as wrappers for `socket.py` (e.g., `Lib/socketserver.py`, `Lib/http/server.py`, `Lib/http/client.py`, `Lib/ftplib.py`, etc.). If you are thinking about using a Python library other than `Lib/socket.py`, `Lib/sys.py` or `Lib/os.py`, please **ask us first**!

Server

The server will be a Python script, named **`server.py`**, executed through the Windows command line interface. The current working directory of the server (i.e., the directory from where the server.py script is executed) will be used as the directory where files will be stored and served from. As your program needs to have write access to said directory, please make sure that, in your Windows command prompt window, you first change directory to someplace where you can store files, then execute your python script from there.

Your server should receive, as its single command line argument, its port number; that is, the server should be executed like so:

```
python server.py <port number>
```

For example, assuming that `M:` is a drive where you have full write access:

```
C:\Users\me> M:
M:\> cd some_dir
M:\some_dir> python server.py 6789
```

On startup, your server should create a TCP server socket, bind it to the user-defined port number (for the hostname use either "0.0.0.0" or an empty string, so as to bind to all available network interfaces on your host), report success (i.e., print a single-line message with its IP address and port number on the console and the text "`server up and running`"), and wait for client connections. For every incoming connection (as returned by `socket.accept()`) it should read and parse the request from the client, serve it accordingly, close the connection, report on its outcome (more on this shortly), and loop back to waiting for the next client connection.

Your server should be able to handle three types of requests:

- Uploading of a file: The client request should include, as a minimum, the request type and the filename to be used on the server side, and the data of the file. The server should then create the file (in exclusive creation, binary mode) and copy the data sent by the client from the socket to the file. To avoid accidents, the server should deny overwriting existing files.
- Downloading of a file: The client request should include, as a minimum, the request type and the filename of the file to be downloaded. The server should then open the file (in binary mode) and copy its data to the client through the socket.
- Listing of 1st-level directory contents: The client request should indicate the request type. The server should then construct a list of the names of files/directories at the top level of its current working directory (e.g., using `os.listdir()`) and return it to the client over the socket. For the needs of this project, your server should not need to handle subdirectories; i.e., all files served and/or entries returned by `os.listdir()` should be those at the top level of the server's current working directory.

In every case, the server should report (i.e., print on the console) information for every request after its processing is over. This report should be a single line including, at the very least, the IP address and port number of the client, information on the request itself (type and filename, as appropriate) and its status (success/failure). For failures, the report should also include an informative message indicating the type of error. Last, the server should also print informative messages for any other types of errors encountered throughout its execution (again, please only print a single line per error).

Client

The client will be a Python script, named **`client.py`**, executed through the Windows command line interface and receiving its arguments as command line arguments. The first argument should be the address of the server (hostname or IP address) and the second argument should be the server's port number. The next argument should be one of "`put`", "`get`" or "`list`"; these signify that the client wishes to send or receive a file, or request a directory listing, respectively. For "`put`" and "`get`" there should then be one more argument with the name of the file to upload/download respectively. That is, the client should be executed like so:

```
python client.py <hostname> <port> <put filename|get filename|list>
```

For example:

```
M:\some_dir> python client.py localhost 6789 put test1.txt
M:\some_dir> python client.py localhost 6789 get test2.txt
M:\some_dir> python client.py localhost 6789 list
```

The client should parse its command line arguments and decide what operation is requested. It should then create a client socket, connect to the server defined in the command line, construct and send an appropriate request message, receive the server's response, process it, and finally close the connection. The processing of requests will depend on the request type:

- Upload ("`put`") request: The client should, at the very least, open (in binary mode) the local file defined on the command line, read its data, send it to the server through the socket, and finally close the connection.
- Download ("`get`") request: The client should, at the very least, create the local file defined on the command line (in exclusive binary mode), read the data sent by the server, store it in the file, and finally close the connection. To avoid accidents, the client should deny overwriting existing files.
- Listing ("`list`") request: the client should, at the very least, send an appropriate request message, receive the listing from the server, print it on the screen one file per line, and finally close the connection.

In every case, the client should report information for every request; this report should be a single line of text including, at the very least, the IP and port number of the server, information on the request itself (type and filename, as appropriate), and its status (success/failure). For failures, the report should also include an informative message indicating the type of error (within the same single line).

Miscellanea

Your client request and server response messages may need to include additional fields to those outlined above. The design of the application-level protocol (i.e., the types and formats of exchanged messages, and the exact semantics and order in which these messages are exchanged) is left up to you and is indeed a major component in the marking scheme of this assessed exercise.

As several pieces of the logic will be shared between client and server, you should try to abstract out the common pieces -- i.e., define functions in a shared module, imported and used by both client and server. For example:

- `send_file(socket, filename)`: Opens the file with the given filename and sends its data over the network through the provided socket.
- `recv_file(socket, filename)`: Creates the file with the given filename and stores into it data received from the provided socket.
- `send_listing(socket)`: Generates and sends the directory listing from the server to the client via the provided socket.
- `recv_listing(socket)`: Receives the listing from the server via the provided socket and prints it on screen.

These functions could then be used by both sides; e.g., for a "`put`" request, the client will use `send_file(…)` while the server will use `recv_file(…)`; vice-versa for a "`get`" request; the functions for the listings could internally also make use of the file functions; etc.

Please make sure that your code is well formatted and documented, and that an appropriate function/variable naming scheme has been used. You won't be assessed on the quality of your Python code per se, but a well written implementation is surely easier to debug (and mark).

What to submit

For this assessed exercise, **you should work on your own**. Submit a single zip file via the course's Moodle page having your matriculation ID and student name as the name of the zip file (**e.g., 1234567M.zip**). The zip file should contain your Python source code files. **The submission deadline is week 6, Friday, October 30th at 16:30.**

Code Explanation/Demo

Your submitted code will have to be explained/demonstrated to your tutors/demonstrators during week 7/week 8 lab sessions. Thus, you will be allocated 5-10 minutes to:

(a) share your screen (i.e. the Moodle tab on your browser) through MS Teams and download your submission so the tutor/demonstrator can verify that you will demo the code you submitted.

(b) Run an example test of your code through a command prompt/shell. **The set of tests will be provided in advance in a separate document on Moodle.**

(c) Answer 2-3 questions related to your code/program logic. **Questions will not be provided in advance.**

(d) Explain the logic behind your application design.

During these allocated 5-10 minutes you will be expected to have **only** your MS Teams, the Moodle tab on your browser, a command prompt/shell, and your code editor. **You will be allocated in advance a specific 5-10 min slot during either in week 7 or week 8. Allocation of your 5-10min slot will be announced during week 6 labs by your demonstrators.**

NO SHOW POLICY:

- **If you do not show in your pre-allocated 5-10 min slot you will lose** 3**0% of the final mark regardless of the quality in your submitted code**. For example, if your code got 35 out of 70 marks (i.e., is half-functional) but you did not demo it, all the 30 marks will be deducted, resulting into an overall grade of 35 out of 100 as the final mark for exercise 1.
- **Exceptional circumstances**: If you are not able to attend your pre-allocated 5-10min slot due to illness or a last-minute unexpected circumstance you will have to inform your tutor/demonstrator in advance (preferably 2 days in advance) and complete the appropriate Good Cause form.

How Exercise 1 will be marked

Following timely submission on Moodle in synergy with the code demo, the exercise will be given a numerical mark,
between 0 (no submission) and 100 (perfect in every way). These numerical marks will then be converted to a band (A1, A2, etc.).

The marking scheme is given below:

- 70 marks for the implementation:
    - 15 marks for the implementation of the "`list`" request type and 25 marks for each of "`put`"/"`get`" request types, broken down as follows:
        - 9 marks for handling the intricacies of TCP communication – i.e., that data is streamed from the source to the destination and hence data sent via a single `send()`/`sendall()` call may be fragmented and received across several sequential `recv()` calls, or data sent via multiple `send()`/`sendall()` calls may be collated and returned through a single `recv()` call. *(All request types)*
        - 3 marks for handling of connection failures mid-way through the protocol. *(All request types)*
        - 2 marks for appropriate logging/reporting. *(All request types)*
        - 1 mark for parsing of command line arguments. *(All request types)*
        - 5 marks for correct handling/transferring of binary data (binary transfer, byte ordering, etc.). *(Only for "`put`"/"`get`" requests)*
        - 5 marks for support for stability/security features such as very large files, 0-sized files, no overwriting of existing files, very long filenames, etc. *(Only for "`put`"/"`get`" requests)*
    - 5 marks for appropriate structure of your code (functions, minimal repetition of code, informative but not excessive comments, etc.).

- 30 marks for the demo:
    - **10 marks**: functional demo running the test asked by your tutor/demonstrator. 5 marks if part of the test works; 0 marks if it doesn't work.
    - **10 marks**: you answer all questions raised by your tutor/demonstrator; (1-5 marks if you partially answer, 5-7 if your answer is satisfactory but not complete, 0 marks if you do not provide an answer at all).
    - **10 marks**: Being able to describe the design and the behaviour of your protocol: e.g., exact format of the exchanged messages, their fields and semantics, the order in which these messages are expected to be exchanged etc., as well as reflect on the submitted solution: e.g., what have you learnt in this assessed exercise, is there anything you would have done differently, etc.; (1-5 marks if you partially answer and your response is incomplete, 5-8 if your answer is satisfactory but not complete, 8-10 if your answer is good and close to perfect; 0 marks if you do not provide an answer at all).