Networks and Operating Systems Essentials 2 (NOSE 2)

Adam Tong	Oliver Livingston
2733692T	2756026L
LB01	LB05

After emailing Awais, he allowed us to do the lab together despite being in separate lab groups. We will be attending each others lab sessions in order to ensure we are present for evaluation.

Assessed Exercise 1: Networking Report

Introduction

As stated in the specification sheet for this assessed exercise, the application consists of two python scripts, a server script called server.py, and a client script called client.py. The purpose of these two files are to simulate a client-server architecture, using TCP connection and the POSIX sockets. This makes use of the python socket library to allow both the scripts to communicate. This report will include the design decisions, protocol specifications and the challenges we encountered during the process of developing the architecture.

Design of the Application-Level Protocol

For this task, we needed three scripts, server.py, client.py and ae1.py. These contain the code for the server-side, client-side and the defined functions/processes respectively. We decided to store all the sending and receiving functions in a separate file, so that both the server and client scripts can access these functions abstractly.

As referenced above in the introduction we need to connect the client to the server, and allow the user to send requests. The connection requires the server to be initialised:

```
#initialising the socket for the server and the hostname for connection.
srv_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host_name = "0.0.0.0"
```

And then needs to have the server wait for a client to connect:

```
try:
    srv_sock.bind((host_name, int(sys.argv[1])))
```

```
    srv_sock.listen()
    print("Server listening on port", sys.argv[1])
```

Much like the server, the client will initialise it's socket for connection and data transmission, and requires a specified server address and a port number so it can connect to the server:

```
#initialising the socket for the client and specify the server address and port number for connection.
cli_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
srv_addr = (sys.argv[1], int(sys.argv[2]))
```

After the client socket has been created, the client sends a connection request to the server:

```
#printing messages to show the user the connection process and .connect line to connect to the server.
try:
    print("Connecting to " + str(srv_addr) + "...")
    cli_sock.connect(srv_addr)
    print("Connected.")
```

Three request were curated in order to meet the assessed specification; the "put" command to upload files from the client to the server, the "get" command to download files from the server-side to the client-side, and a "list" command to retrieve directory listings.

In line 7 of client.py, there is a list of commands that the user chooses from, and a "try" condition is used in line 35 to specify the task and avoid running into errors during runtime.

For each request, the client must use the specified command keywords and the server must receive the command in order to either, receive the file from the client, send a file to the client, or allow the client to view the directory list.

Success/Failure

For each client-server interaction, a single line will print through in the form of **<Server/Client:(IP, Port Number)><Filename><Command><SUCCESS/FAIL>.**
This line will print through on both the server-side and the client-side. We use a string variable called report to handle this by adding all the relevant info to it as we sequentially move through the programs.

If the code runs into any errors, an informative error message will print. This is largely done by using Try/Except statements within the code to control the flow of the program and handle any unexpected errors.

"Put" protocol

The put protocol is sent from the client side in the form of:

<Request Type> <Filename>

where the request type is the “put” keyword and the filename is the file attempted to being sent over.

This means that the client requests an upload to the server. The message order goes:

1. **The client initiates a connection to the server.**
2. **The client sends a request message to send file.**
3. **The server processes the request and sends a response message.**
4. **The client receives and processes the response, sending the file.**
5. **The connection is closed.**

For our solution, in the server.py we have 2 arguments when we start the server:

‘python server.py 1069’

The first argument is the python script and the second argument is the port number.

In the client.py we have 5 arguments when we start the server:

‘python client.py localhost 1069 put "nose.jfif"’

The first argument is the client python script, the second argument is the server address, the third is the port number, the fourth is the command keyword and the fifth final argument is the filename. The server address and the port number allow the client connect, and then the command can be carried out.

In order to carry out a send_file, we split the package we send into bytes by encoding it. This allows it to be passed from the client socket to the server socket. This operation is carried out by the “send_file()” function in “ae1.py”.

On the other side, the server runs recv_file(), which allows the server-side to receive the file and decode it. We set the socket to receive 1024 bits at a time to avoid bottlenecking and will be able to receive larger files from fragmentation. The function for this is stored in the “ae1.py” script, line 30, and is accessed by “server.py” in this instance. This means that the file will be created in the server directory, and will send confirmation of receiving the file.

In summary, the `put` method effectively sends a file to the server through a socket connection, ensuring data integrity and successful transmission. It provides a robust mechanism for uploading files to the server, enhancing the functionality of our networked file service application.

“Get “ Protocol

The get protocol is sent from the client side in the form of:

<Request Type> <Filename>

where the request type is the “get” keyword and the filename is the file attempted to being sent over.

This means that the client requests an upload to the server. The message order goes:

1. **The client initiates a connection to the server.**
2. **The client sends a request message, requesting the file.**
3. **The server processes the request and sends a response message.**
4. **The client receives and processes the response, receiving the file.**
5. **The connection is closed.**

For our solution, in the server.py we have 2 arguments when we start the server:

‘python server.py 1069’

The first argument is the python script and the second argument is the port number.

In the client.py we have 5 arguments when we start the server:

‘python client.py localhost 1069 get "myText.txt"’

The first argument is the client python script, the second argument is the server address, the third is the port number, the fourth is the command keyword and the fifth final argument is the filename. The server address and the port number allow the client connect, and then the command can be carried out. In this case it is to download a text file from the server.

In order to do this, the client sends a get request to the server, which runs a send file function from the ae1.py file, and then the client receives the encoded file, runs the decode and stores the file in the client directory.

The code used to send the request is in the server.py script, line 35 with the try condition.

The server then receives the data from the client in line 17, within the while loop, using the .recv in line 24.

The server recognises that it is a get command and runs the send file, which is the same function used by the put function in the previous protocol, cutting down code by reusing functions.

On the other-side, the client performs the receive file, which is also the same function used by the put function in the previous protocol, again cutting down code making it more abstract.

In conclusion, the `get` method allows the client to download files from the server with a reliable mechanism for data transfer. It ensures the client receives the requested file, providing a crucial component for our networked file service application, enhancing file retrieval capabilities.

List Protocol

The list protocol provides the client with a directory list, and does this by sending a list request to the server, and the server sends a list of files within that directory back, before closing the connection.

When the client sends a `list` request to the server, the client creates a message indicating that it wants a directory listing. This message is then sent to the server over a network connection. This is done in the try condition of line 19

This check is necessary, as a list command from the client side only has 4 arguments rather than the typical file, as it is not requesting or sending any files. This means that no file is needed in the command.

The server, upon receiving the `list` request from the client, processes the request. It uses Python's `os.listdir()` to obtain a list of files and directories present in its current working directory (the directory where files are stored). This listing includes the names of files and directories at the top level of the server's directory structure. The code is stored within line 52.

Once the server obtains the directory listing, it prepares a response message that contains this listing. The server sends this message back to the client through the network connection. The server code for this lies on line 45.

The client, after sending the `list` request, waits to receive the server's response. Upon reception, the client reads the response message, which includes the names of files and directories in the server's directory. The code for this is in line 40 in ‘client.py’.

The client then processes the response message, extracts the list of files and directories, and displays this list to the user. This list is typically printed on the client's console or displayed in a user-friendly manner. The called function is in line 63.

After this the client, much like the other protocols, will close after the operation has been carried out.
