Computer Architecture and Network Systems

Problem Set, Unit B3

Sockets

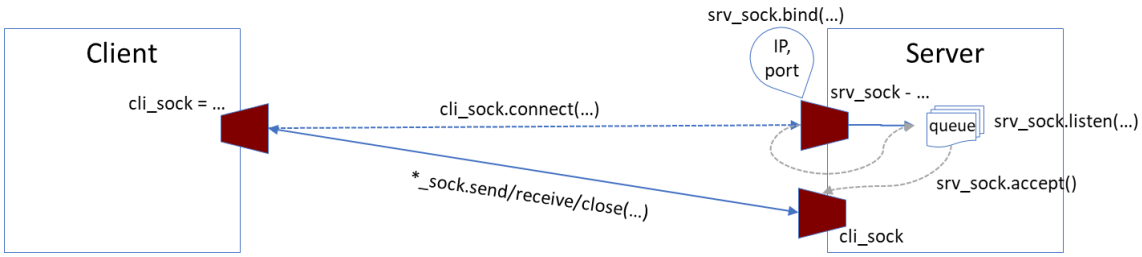
Setup
Computer based exercises

The aim of this lab is to familiarise you with the Python socket library in the context of building a simple "chat" application using TCP/IP.

POSIX Sockets

An Internet socket is an abstract representation for the local endpoint of a network connection. Berkeley sockets is a UNIX API for Internet sockets, coined after the first implementation of sockets appeared in the 4.2BSD Unix distribution (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 and others.

The basic workflow for setting up a TCP connection to a remote host using POSIX sockets consists of the following steps. Please consult https://docs.python.org/3/library/socket.html for the various methods.



Server software:

```
import socket
import sys

# Create socket
srv_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Register the socket with the OS (defines IP and port no. of
the endpoint)
srv_sock.bind((local hostname, port number))

# Create a queue for incoming connection requests
srv_sock.listen(new connection requests queue length)

while True:
      # Block until a new connection request arrives
      # Returns a new socket connected to the client
      # and the client's address
      cli_sock, cli_addr = srv_sock.accept()

      while some condition:
            # Receive request
            request = cli_sock.recv(request msg max length)

            # Process request, if applicable
            # …

            # Send response, if applicable
            cli_sock.sendall(response message)

      # Close/disconnect client socket
      cli_sock.close()
```

Client software:

```
import socket
import sys

# Create socket
cli_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect socket to server
cli_sock.connect((remote hostname, port number))

while some condition:
      # Prepare request message
      # …

      # Send request
      cli_sock.sendall(request message)

      # Receive response, if any
      cli_sock.recv(response message max length)

      # Process response, if applicable
      # …

# Close/disconnect socket
cli_sock.close()
```

For example, here is some code that implements a simple protocol (the client sends a string to the server, who then prints it on its screen).

myserver.py:

```
import socket
import sys

srv_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
srv_sock.bind(("", int(sys.argv[1])))
srv_sock.listen(5)
while True:
   cli_sock, cli_addr = srv_sock.accept()
   request = cli_sock.recv(1024)
   print(str(cli_addr) + ": " + request.decode('utf-8'))
   cli_sock.close()
```

Execute it like so, from Anaconda 3's command prompt
(or any other where python is available):

```
python myserver.py 2000
```

where 2000 is just a port number in the range [1024, 65535]. If the port number you choose is already in use, the code will terminate with an exception (OSError: [Errno 98] Address already in use). In that case, try executing your server again with a different port number in the above range. When executed properly, the server should wait (block) for new requests. On reception of a request, it will print the client's address (IP address, port number of client port) and the string the client sent. You can terminate the server by hitting Ctrl+C on your keyboard.

myclient.py:

```
import socket
import sys

cli_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
cli_sock.connect((sys.argv[1], int(sys.argv[2])))
cli_sock.sendall(sys.argv[3].encode('utf-8'))
cli_sock.close()
```

Execute it like so, from Anaconda 3's command prompt (*you will need to open up a second window in which to execute your client*):

```
python myclient.py localhost 2000 "this is my message"
```

where localhost is the hostname where the server is running ('localhost' refers to the same host as the one the client program is executed), 2000 is the port number where the server is waiting for connections (should much whatever you used for your server above), and the double-quoted string is the message to be sent (needs to be in quotes to force it to be a single argument in sys.argv.

Note: The above code snippets are provided only as a starting point. Obviously, they lack proper input parsing, exception handling, program termination, etc. More notably, they make various network-related assumptions; e.g., that incoming messages will always be less than 1024 bytes, or that all of the data sent from the client will have arrived at the server when the latter executes its socket.recv() method, and so on. In the general case, you should make sure that your code doesn't make these same assumptions.

Lab Tasks

In this lab, you are asked to amend/augment the above code to implement a simple TCP "chat" application. The application will allow two users to send and receive text messages over the internet. In this case, one of the users will be executing a server while the other a client, but the functionality other than that should be similar.

The client should receive the remote host name and port number **on the command line** *(this is where previous tutorial comes in)*, connect to the remote server, and then exchange messages – i.e., read data from the keyboard and send it to the server whenever a newline is detected (Enter key pressed), then read data from the server and print it on the screen, in this order – until either the local user enters the keyword "EXIT" on a line of its own (no leading spaces), the server closes the connection, or a network error occurs. In the latter case, the client should print a diagnostic message, while in any other case it should just exit.

The server should, in turn, receive the port number to bind to on the command line (for the hostname use either "0.0.0.0" or an empty string), create its endpoint, and then exchange messages with clients. Specifically, for each incoming connection it will enter in a loop where it will read data sent from the client and print it on the screen, then read data from the keyboard and send it to the client, in this order. If the local user enters the keyword "EXIT" on a line of its own, or the client closes the connection, or a network error occurs, the server should close the client socket and return to

waiting for the next incoming connection. Network errors should also be accompanied by an appropriate diagnostic message printed on the server screen.

What to Submit

Nothing… However, before leaving the lab room, please make sure that (a) you understand every single line of the example code, and (b) you have a working implementation of your chat application as **it will form the basis of your 2nd assessed exercise.**