COMP3234B Computer and Communication Networks

Lab 1: Socket Programming

Introduction

In this lab, we will practice Python socket programming. We will first try out the example client and server programs in the lecture, and then implement a simple TCP-based client/server game.

Python Socket Programming

Go to https://www.python.org/downloads/ to download and install the latest Python release on your operating system.

To use sockets in your Python programs, you must include the Socket Module in your programs, which contains Low-level networking interface (close to the BSD API):

```
from socket import *
```

or

```
import socket
```

(see their differences on page 14 of lecture slides 3_SocketProgramming_COMP3234B_s2024.pdf)

The following is a list of often used Python socket APIs for your reference (see more at https://docs.python.org/3/library/socket.html):

s = socket.socket (socket_family, socket_type, protocol)	create a socket: • socket_family : AF_INET (IPV4) or AF_INET6 (IPV6) (commonly used two; see others at https://docs.python.org/3/library/socket.html) • socket_type: SOCK_STREAM (TCP), SOCK_DGRAM (UDP) (commonly used two; more at https://docs.python.org/3/library/socket.html) • protocol: This is defaulting to 0

Server socket functions:

s.bind(address)	bind address to socket: • address is (hostname or IP address, port number) for IPV4 (AF_INET); • address is (host, port, flowinfo, scopeid) for IPV6 (AF_INET6)
s.listen([backlog])	used by TCP server program; set up and start TCP connection listener:

	• backlog specifies the max. no. of incoming connection requests that can be queued while waiting for server to accept them, which is optional
s.accept()	used by TCP server program to accept TCP client connection: • return value: a pair, (conn, address), where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection

Client socket functions:

s. connect(address)	establish a connection to a server socket at address, which is waiting at accept()

Socket functions that both client and server can use:

s.recv(bufsize[, flags])	receive TCP stream data from the socket: • bufsize: maximum amount of data to be received at once • flags: optional; see more at https://man7.org/linux/man-pages/man2/recv.2.html • return value: a bytes object representing the data received
s.send(bytes[, flags])	send TCP stream data to the socket: • bytes: the data object to send • flags: optional, same meaning as for recv() above • return value: the number of bytes sent
s.recvfrom(bufsize[, flags])	receive UDP datagrams from the socket: : • bufsize: maximum amount of data to be received at once • flags: optional, same meaning as for recv() above • return value: a pair, (bytes, address), where bytes is an object representing the data received and address is the address of the socket sending the data
s.sendto(bytes[, flags], address)	send UDP datagrams to the socket: • bytes: the data object to send • flags: optional, same meaning as for recv() above • address: address of the destination socket • return value: the number of bytes sent
s.close()	close the socket
socket.gethostname()	return a string containing the hostname of the machine where the current program is executing.
s.getsockname()	return the current socket's own address: • return value: address in a format according to the socket's address family

Run example programs

We have provided source code of a few examples for socket programming in socketprog_examples.zip. Please try them out as follows.

Example 1 (TCPSocket-1): This is the TCP server (sequential) and client example in the lecture slides.

- **Launch one terminal** and switch to the directory of TCPSocket-1. Run the server program as follows:

```
python3 TCPServer.py
```

You will see the prompt in the terminal "The server is ready to receive", which means the server is listening.

- **Launch another terminal** and switch to the directory of TCPSocket-1. Run the client program as follows:

```
python3 TCPClient.py
```

You will see the prompt "Input a lowercase sentence:". You can enter such a sentence and press "enter". Then you will see an uppercase sentence returned from the server.

Example 2 (TCPSocket-2): This is equivalent implementation of the TCP server (sequential) and client example in the lecture slides, where instead of using "from socket import *" to import APIs in the socket module, we use "import socket". Compare the difference between the programs in Example 1 and Example 2.

Run the example following the same steps as given in Example 1. (You can use control+C to kill a running server.)

Example 3 (TCPSocket-3): This is another implementation of the TCP server (sequential) and client example in the lecture slides, where we add a number of exception handling codes. Read more about Python error and exception handling at https://docs.python.org/3/tutorial/errors.html, exception socket.error at https://docs.python.org/3/library/socket.html#socket.error, Python sys module at https://docs.python.org/3/library/sys.html .

Run the example following the steps given in Example 1. When you see the prompt "Input a lowercase sentence:", try using "control+c" instead of entering a sentence; then you will see error prompt on both terminals running client and server programs.

Example 4 (TCPSocket-4): This is the TCP server (concurrent) and client example in the lecture slides, where the server can handle concurrent connections through threading.

- **Launch one terminal** and switch to the directory of TCPSocket-4. Run the server program as follows:

```
python3 TCPServer.py
```

You will see the prompt in the terminal "The server is ready to receive", which means the server is listening.

- **Launch the second terminal** and switch to the directory of TCPSocket-4. Run the client program as follows:

```
python3 TCPClient.py
```

- **Launch the third terminal** and switch to the directory of TCPSocket-4. Run the client program as follows:

```
python3 TCPClient.py
```

Now two connections are set up with the same server. You can enter sentences on the second and third terminals and communicate with the server concurrently.

Example 5 (TCPSocket-5): This is an equivalent implementation of TCP server (concurrent) and client as Example 4, where we use the Thread class to implement multi-threading in the server program (https://docs.python.org/3/library/threading.html#threading.Thread). Read more about multithreading programming with Python at https://www.tutorialspoint.com/python/python_multithreading.htm.

Besides, the server program runs starting from the following code:

```
if __name__ == '__main__':
        server = ServerMain()
        server.server_run()
```

The purpose of the line of code if __name__ == '__main__' is to tell whether the current module is read directly by the Python interpreter, i.e., whether your program is run as the main program (read more at https://stackoverflow.com/questions/419163/what-does-if-name-main-do). ServerMain() is to create an instance of the ServerMain class, and then we run the method server_run() defined in the ServerMain class.

Test the programs following steps given in Example 4.

Example 6 (UDPSocket): This is the UDP server and client example in the lecture slides.

- **Launch one terminal** and switch to the directory of UDPSocket. Run the server program as follows:

```
python3 UDPServer.py
```

You will see the prompt in the terminal “The server is ready to receive”.

- **Launch another terminal** and switch to the directory of UDPSocket. Run the client program as follows:

```
python3 UDPClient.py
```

You will see the prompt “Input a lowercase sentence:”. You can enter such a sentence and press “enter”. Then you will see an uppercase sentence returned from the server.

Lab Exercise: Simple TCP-based Client/Server Number Guessing Game

We now implement a simple number guessing game, where the server generates a secret integer number and the client sends guesses to the server using TCP. The server replies "Too high" or "Too low" until the client successfully guesses the number.

Step 1: Download **lab1_materials.zip** from Moodle. Unzip it and you will find two files provided: **server.py** and **client.py**.

Step 2: Open **client.py** using a text editor. **client.py** contains the complete implementation of the client program. Study the client program carefully and you will learn from its code to complete the server program.

a. The client is to be started by command "python3 client.py". Once started, it creates a TCP connection to "localhost" on port 12000 in lines 3-11. You may get a ConnectionRefusedError if you run the script now because the server program is not running. In **Step 3**, we will implement the corresponding server code.

b. After establishing the TCP connection, the client runs the game logic enclosed in a try-finally block. This structure ensures that the TCP connection is properly closed in case of errors. More details at https://docs.python.org/3/tutorial/errors.html#defining-clean-up-actions.

c. The client side game logic starts with reading a guess from the player using input(). We add a new line character '\n' to indicate the termination of a guess. The guess is then sent to the server via the TCP connection. To receive the complete response from the server, we use a while loop to keep recv() from the TCP connection, until a new line character '\n' is received. The script then presents the received response to the player and breaks the while loop if the guess is correct.

Step 3: Open **server.py** using a text editor and you will find that it provides a sketch of the server program. Complete the ServerMain class following the hints given as "#....".

a. In the server_run() method, set up a TCP socket that listens on port 12000.

b. Inside the while loop in the server_run() method, accept one connection and launch a ServerThread to serve the game. You can refer to **Example 5 (TCPSocket-5)**. Remember to pass both the connection socket and the secret number (the random number that the server generates) to the constructor of ServerThread.

Step 4: Edit **server.py** and complete the game logic in the run() method of ServerThread.

a. It should include a while loop to repeatedly read player guesses. Inside the while loop, read the player's guess from the TCP connection until a new line character '\n' is received. Then convert the guess from a string into an integer number using int() (https://docs.python.org/3/library/functions.html#int).

b. Compare the player's guess with the secret number on the server and reply "Too high\n" or "Too low\n" if the guess is wrong. We always append a new line character '\n' to indicate the end of a response so the client can start asking the player for the next guess.

c. If the guess is correct, reply "You win!\n" and break the while loop.

d. In case of any error, close the TCP connection.

Step 5: test your programs as follows:

- **Launch one terminal** and switch to the directory of Lab1. Run the server program as follows:

```
python3 server.py
```

- **Launch the second terminal** and switch to the directory of Lab1. Run the client program as follows:

```
python3 client.py
```

Here is a sample output when running the application on the same machine. The server side output is optional.

Server program:

```
> python server.py
Game started! The secret is 37
Received guess: 50
Received guess: 25
Received guess: 37
```

Client program:

```
> python client.py
Input a guess: 50
Too high

Input a guess: 25
Too low

Input a guess: 37
You win!
```

Submission:

You should submit the following files:
(1) server.py
(2) client.py
Please compress the above files into a lab1-yourUID.zip file and submit it on Moodle before 23:59 Wednesday Jan. 31, 2024:

(1) Login Moodle.
(2) Find “Labs” in the left column and click “Lab 1”.
(3) Click “Add submission”, browse your .zip file and save it. Done.
(4) You will receive an automatic confirmation email, if the submission was successful.
(5) You can “Edit submission” to your already submitted file, but ONLY before the deadline.