

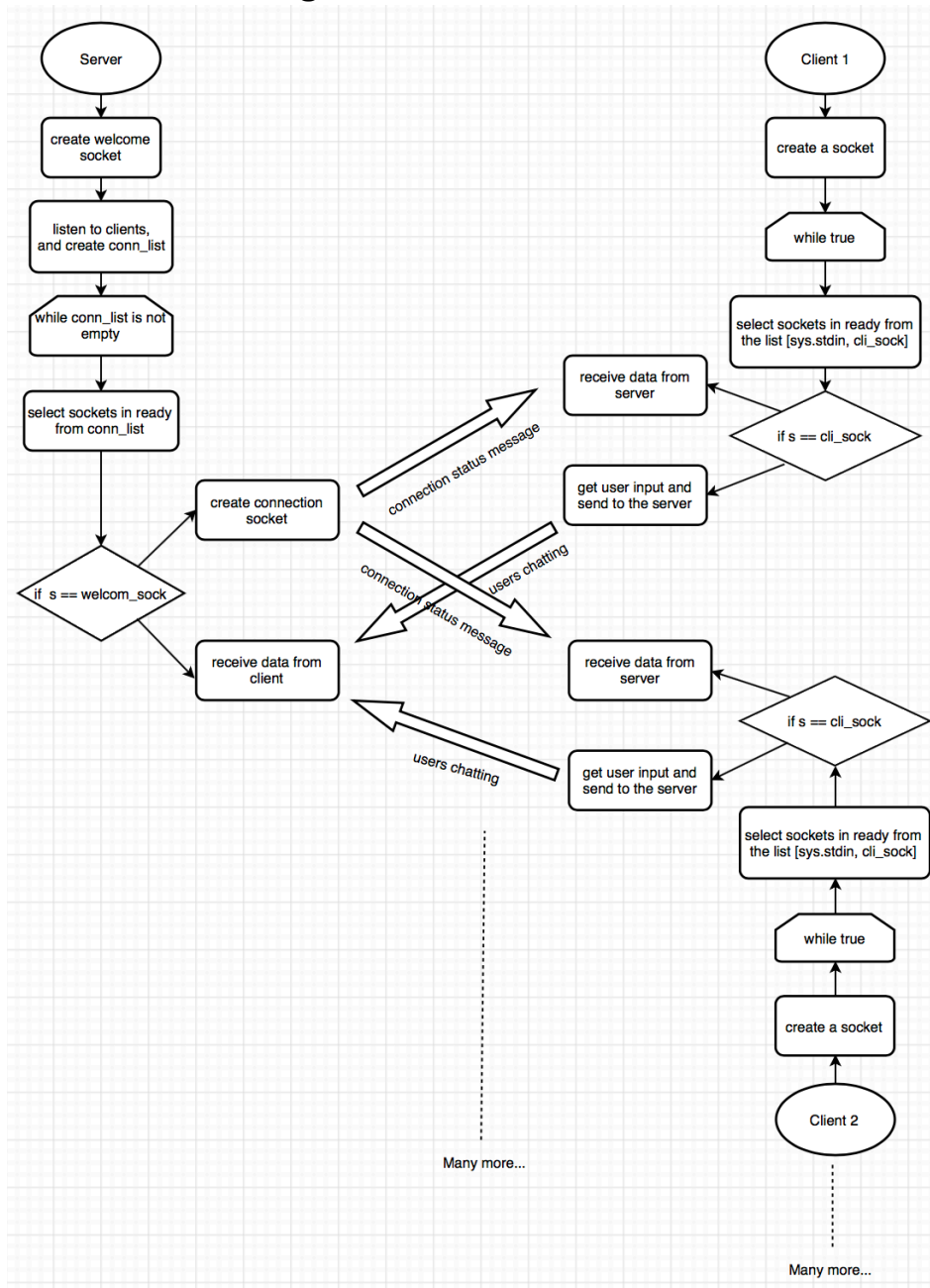
[Report] Project 1

2015121088 Soyoung Kang

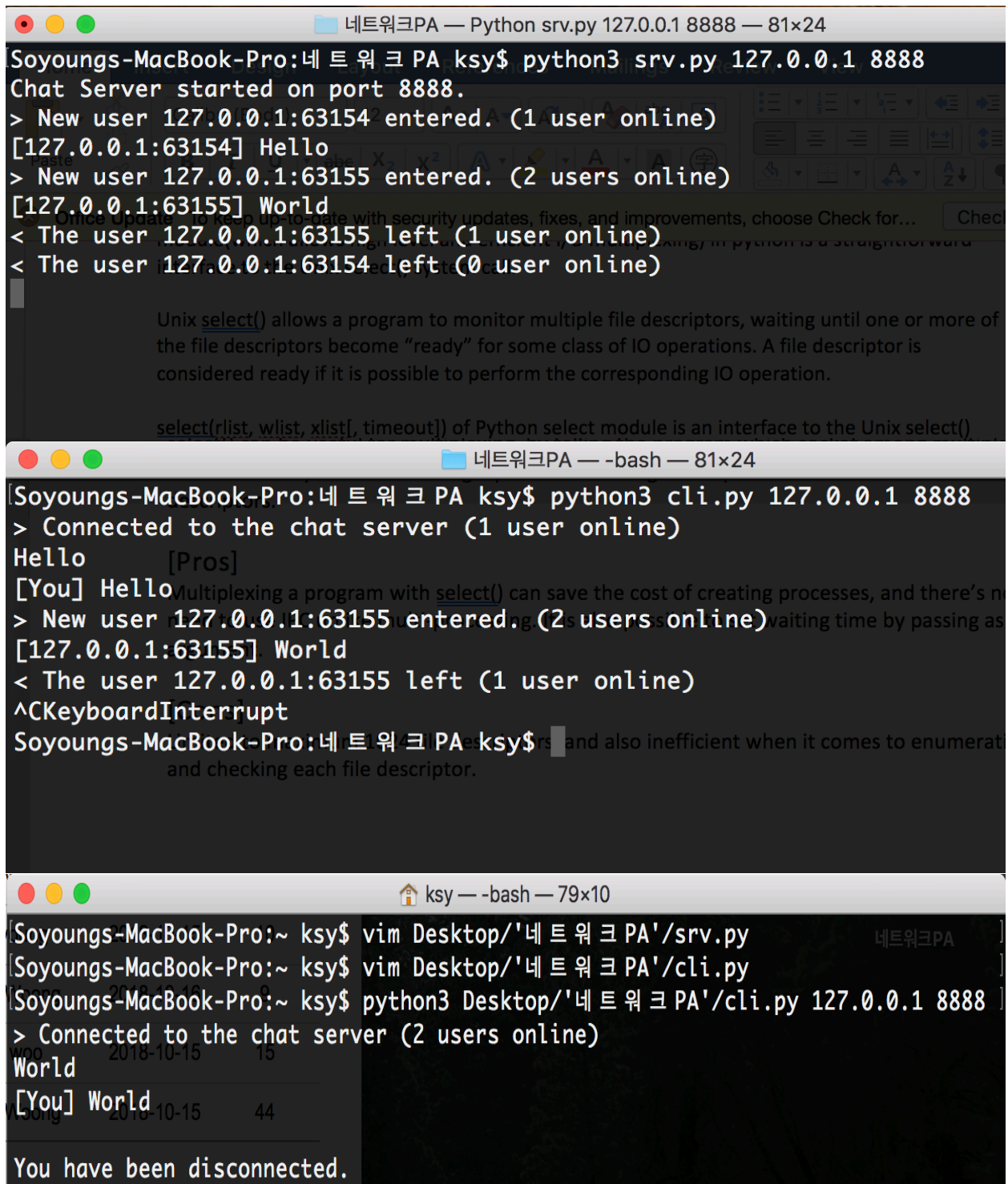
Introduction

This is a multi-user real time chatting application based on python language and its select module. I implemented the entire server side code and updated skeleton client code so that the program processes multiple users with a single server by multiplexing.

Flow chart or Diagram



Snapshots



The image displays three sequential terminal window snapshots from a macOS environment, showing the development and testing of a chat server and client.

Snapshot 1: Server Setup
The terminal window is titled "네트워크PA — Python srv.py 127.0.0.1 8888 — 81x24". The user runs `python3 srv.py 127.0.0.1 8888`. The output shows the chat server starting on port 8888. It logs two users connecting (127.0.0.1:63154 and 127.0.0.1:63155) and their messages ("Hello" and "World"). It also logs two users disconnecting. A background window titled "Once Update" is visible.

Snapshot 2: Client Interaction
The terminal window is titled "네트워크PA — -bash — 81x24". The user runs `python3 cli.py 127.0.0.1 8888`. The output shows the client connecting to the chat server. It displays the server's response "Hello" and the user's input "Hello". It also shows the user sending "World" and the server responding "World". The user then presses Ctrl-C, resulting in a "KeyboardInterrupt" message. A background window titled "Multiplexing a program with select()" is visible.

Snapshot 3: File Editing and Re-run
The terminal window is titled "ksy — -bash — 79x10". The user runs `vim Desktop/'네트워크PA'/srv.py` and `vim Desktop/'네트워크PA'/cli.py`. Then, the user runs `python3 Desktop/'네트워크PA'/cli.py 127.0.0.1 8888`. The output shows the client connecting to the chat server. It displays the server's response "World" and the user's input "World". The user then presses Ctrl-C, resulting in a "KeyboardInterrupt" message. A background window titled "You have been disconnected." is visible.

Logical explanations block by block in detail

[srv.py]

```
import socket, sys, select
2
welcome_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
hosts = sys.argv[1]
ports = int(sys.argv[2])
6
welcome_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
welcome_sock.bind((host, port))
print('Chat Server started on port %s.' % port)
10 while True:
num_cli = 0
welcome_sock.listen()
conn_list = [welcome_sock]
```

Firstly, create a welcome socket `welcome_sock` which sets address as command line argument. Then wait for clients using `listen()`, whilst making a list of connections called `conn_list`, initializing it to the `welcome_sock`.

```
# when there exist connection requests
while conn_list:
17 try:
18     sr, sw, se = select.select(conn_list, [], [])
19     for s in sr:
20         # create new connection
21         if s == welcome_sock:
22             conn_sock, cli_addr = welcome_sock.accept()
23             cli_addr_format = cli_addr[0] + ':' + str(cli_addr[1])
24             num_cli = num_cli + 1
25         conn_list.append(conn_sock)
26         print('KeyboardInterrupt')
27         if num_cli >= 2:
28             sys.stdout.write('> Connected to the chat server (%d users online)' % num_cli)
29         else:
30             sys.stdout.write('> Connected to the chat server (%d user online)' % num_cli)
31             print('KeyboardInterrupt: ', e.message)
32             if num_cli >= 2:
33                 msg = '> New user %s entered. (%d users online)' % (cli_addr_format, num_cli)
34             else:
35                 msg = '> New user %s entered. (%d user online)' % (cli_addr_format, num_cli)
36             print(msg)
37
38         # send to other connected users
39         for c in conn_list:
40             if c != conn_sock and c != welcome_sock:
41                 c.send(msg.encode())
```

While `conn_list` is not empty, call `select.select()` to take client sockets that are ready to connect. For each client socket in ready, create a connection socket `conn_sock` by calling `accept()` on `welcome_sock` if that is newly connecting socket. Add this newly connected `conn_sock` into `conn_list` so that the server can keep tabs on what it's sending. Send the connection status including the total number of users online to both incoming client socket and the rest of connected client sockets, as well as printing on the server side terminal.

```

9  # existing connection
10 else:
11     data = s.recv(1024).decode()
12     s_addr, = s.getpeername()[0]+'!' + str(s.getpeername()[1])
13     # data received
14     if data:
15         # live data from server
16         msg = '[%s] %s' % (s_addr, data)
17         print(msg)
18         # send to other connected users
19         for c in conn_list:
20             # if c != welcome_sock and c != s:
21             else:
22                 c.send(msg.encode())

```

Otherwise if there already exists a connection socket for this socket in the loop, receive data from it. Make sure to propagate the received data to other clients online since this is a real time multi-user chat application.

```

# disconnect if blank received or keyboard interruption by client
else:
    conn_list.remove(s)
    num_cli = num_cli - 1
    s.close()

if num_cli >= 2:
    msg = '< The user %s left (%d users online)' % (s_addr, num_cli)
else:
    msg = '< The user %s left (%d user online)' % (s_addr, num_cli)
print(msg)
# send to other connected users
for c in conn_list:
    if c != welcome_sock and c != s:
        c.send(msg.encode())

```

Disconnect the connection if the client types in invalid message such as keyboard interruption or blank. Remove the connection socket from conn_list and close it.

```

# terminate the server with keyboard interruption
except KeyboardInterrupt:
    print('KeyboardInterrupt')
    for c in conn_list:
        c.close()
    welcome_sock.close()
    sys.exit()

```

Catch KeyboardInterrupt so that the program terminates with Ctrl+C. Close all the existing connection sockets and welcome_socket in this case.

[cli.py]

```
1 import socket, sys, select entered. (%d user online)' %(cli_addr_format, r
2     print(msg)
3 cli_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
4 srv_host = sys.argv[1]
5 srv_port = int(sys.argv[2])
6     if c != conn_sock and c != welcome_sock:
7 # connect to the server
8 cli_sock.connect((srv_host, srv_port))
```

Create a socket cli_sock that takes a server address as command line argument. Then connect to the server.

```
while True:
    try:
        sr, sw, se = select.select([sys.stdin, cli_sock], [], [])
        for s in sr:
            # receive data from server
            if s == cli_sock:
                data = s.recv(1024).decode()
                if data:
                    print(data)
                else:
                    cli_sock.close()
                    print('Server closed the connection.')
                    sys.exit()
```

Run the loop until the connection disconnects, meaning it keeps detecting transmission between server and itself.

If the current socket being processed by the server is this very socket(cli_sock), implying the server is sending some message to it, then receive data from the server. The received data will be messages with status information. If there is no data being received, which indicates the server closed the connection, then close the client socket and exit the program.

```
# send user input to server
else:
    cli_chat = input()
    # disconnect if input is empty string
    if cli_chat == '':
        cli_sock.close()
        print('You have been disconnected.')
        sys.exit()
    else:
        cli_sock.send(cli_chat.encode())
        print('[You] ' + cli_chat)
```

Otherwise get message as a raw input from user and send it to the server so that the server can propagate this message to other users. If the user enter empty string, then disconnect.

```
except KeyboardInterrupt:
    print('KeyboardInterrupt')
    cli_sock.close()
    sys.exit()
```

Catch KeyboardInterrupt so that the program terminates with Ctrl+C. Close the socket and exit program.

All explanations of the 8 functions in “What are these functions?” slide.

- `socket(socket.AF_INET, socket.SOCK_STREAM)`

Creates socket object. The first parameter indicates this socket is in address family 'AF_INET', and the latter parameter indicates this socket is a SOCK_STREAM type.

- `setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)`

Set the value of the given socket options into specified type and state. The first argument SOL_SOCKET is a level of the socket, and SO_REUSEADDR is an optname which returns True when the socket is bound by an address already in use. The last parameter is the size of buffer.

- `select([], [], [])`

Select module allows high level and efficient I/O multiplexing.

`select(rlist, wlist, xlist[, timeout])` is an interface to the Unix `select()` system call. The first three arguments are sequence of 'waitable objects': either integers representing file descriptors or objects with a parameterless method named `fileno()` returning such an integer:

- `rlist`: wait until ready for reading
- `wlist`: wait until ready for writing
- `xlist`: wait for an exceptional condition

Empty sequences are allowed.

The return value is a triple of lists of objects that are ready: subsets of the first three arguments.

- `bind(("0.0.0.0", 7777))`

Bind the socket to address(parameter). The socket must not already be bound.

- `listen(5)`

Enable server to accept connections. The parameter specifies the number of unaccepted connections that the system will allow before refusing new connections.

- `connect((host, port))`

Connect to a remote socket at the address given as an argument.

- `accept()`

Accepts a connection. The socket must be bound to an address and listening for connections. The return value is 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.

- `close()`

Mark the socket closed. All future operations on the socket object will fail.

What is the function of select()?

[Explanation]

As mentioned in the answer to the previous question, select() function from select module(which allows high level and efficient I/O multiplexing) in python is a straightforward interface to the Unix select() system call.

Unix select() allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become “ready” for some class of IO operations. A file descriptor is considered ready if it is possible to perform the corresponding IO operation.

select(rlist, wlist, xlist[, timeout]) of Python select module is an interface to the Unix select() system call. The first three arguments are sequence of ‘waitable objects’: either integers representing file descriptors or objects with a parameterless method named fileno() returning such an integer:

- rlist: wait until ready for reading
- wlist: wait until ready for writing
- xlist: wait for an exceptional condition

Empty sequences are allowed.

The return value is a triple of lists of objects that are ready: subsets of the first three arguments.

select() can be useful for multiplexing, by telling the program which socket among multiple sockets is ready. It enables a single process to manage multiple clients as a set of file descriptors.

[Pros]

Multiplexing a program with select() can save the cost of creating processes, and unlike multiprocessing, there’s no need to use IPC. It is also possible to set waiting time by passing timeout as an argument.

[Cons]

Limited scale is the major downside. For example in case of codes written in C which adopts file descriptor for select, the size is limited to maximum 1024 file descriptors. Also, it’s quite inefficient when it comes to enumerating and checking each file descriptor.

Reference

Consulted official Python documentations for each module, and the sample python socket implementation on textbook slides as reference.