Socket Programming

Introduction

TA: Cristian Rodriguez

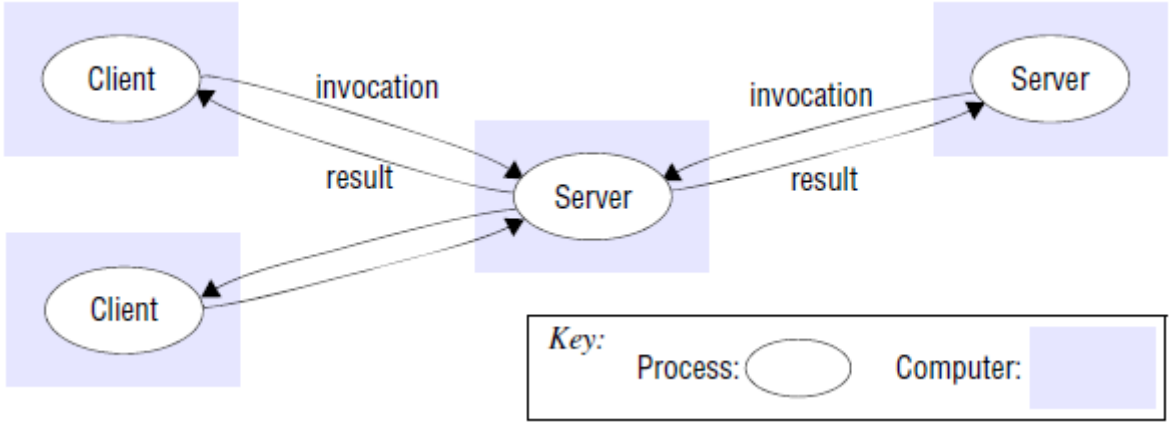
COMP 445 – Winter 2021

Agenda

- Client-Server architecture
- IRC Client installation
- Socket creation - Client
- Socket creation – Server – Blocking
- HTTP 1.0 Client GET
- HTTP 1.0 Client POST
- FTP Client exercise

Client-Server architecture

- Client processes interact with individual server processes in potentially separate host computers in order to access the shared resources that they manage.



Reference: Coulouris, George F., Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. Pearson education, 2005.

Socket creation – Client

```python
import socket

HOST = '192.168.1.1'
PORT = 17517

client = socket.socket()
client.connect((HOST, PORT))
while True:
    message = input("send >> ")
    if message == "exit":
        client.close()
        break
    client.sendall(bytes(message, 'utf-8'))
    data = client.recv(1024)
    print('Received >> ', data.decode("utf-8"))
```

- First, we need to import socket.
- Create the socket specifying its family and type.
  - AF_INET, SOCK_STREAM (default)
  - AF_UNIX, SOCE_DGRAM
- Provide the port number, host name or IP address of the server we want to connect to.
- The socket is ready to send data as bytes encoded with UTF-8.
- Receive data and decode it.

Further details: https://docs.python.org/3/library/socket.html

Socket creation – Server – Blocking

```python
import socket
import threading

HOST = '127.0.0.1'
PORT = 7

def client(socket,address):
    while True:
        message = socket.recv(1024)
        if message:
            print(address, ' >> ', message.decode("utf-8"))
            socket.sendall(message)
        else:
            print(address, ' >> CLOSED ')
            socket.close()
            break

server = socket.socket()
server.bind((HOST, PORT))
print('Waiting for clients...')
server.listen(5)

while True:
    c, addr = server.accept()
    th = threading.Thread(target=client, args=(c,addr))
    th.start()
s.close()
```

- First, we need to import socket and threading.
- Create the socket specifying its family and type.
  - AF_INET, SOCK_STREAM (default)
  - AF_UNIX, SOCE_DGRAM
- Provide the port number, host name or IP address where we want the server listen for incoming connections.
- Listen for incoming connections specifying how big we want the queue.
- Accept connections request and create a new thread to serve the client.

Further details: https://docs.python.org/3/library/socket.html

Socket creation – Server – Blocking

```python
import socket
import threading

HOST = '127.0.0.1'
PORT = 7

def client(socket,address):
    while True:
        message = socket.recv(1024)
        if message:
            print(address, ' >> ', message.decode("utf-8"))
            socket.sendall(message)
        else:
            print(address, ' >> CLOSED ')
            socket.close()
            break

server = socket.socket()
server.bind((HOST, PORT))
print('Waiting for clients...')
server.listen(5)

while True:
    c, addr = server.accept()
    th = threading.Thread(target=client, args=(c,addr))
    th.start()
s.close()
```

- Potential problem?
  - It is blocking so; it may be the case that the server gets blocked waiting for client's data.
  - It is possible to specify a timeout with `socket.settimeout(value)`. Where value specify the time in seconds before timing out. If no value is specified, then the socket is in blocking mode. If zero, then the socket is in non-blocking mode.
- To receive data, it is necessary to specify the buffer size. Normally it is a number power of two.

Further details: https://docs.python.org/3/library/socket.html

Exercise – HTTP 1.0 Client

- RFC 1945 specifies HTTP/1.0 protocol, so let's look how the request looks like. Page 35

```
Request-Line   = Method SP Request-URI SP HTTP-Version CRLF
```

```
Method         = "GET"                ; Section 8.1
               | "HEAD"               ; Section 8.2
               | "POST"               ; Section 8.3
               | extension-method
```

CRLF = \r\n
SP = space

```
Request        = Simple-Request | Full-Request

Simple-Request = "GET" SP Request-URI CRLF

Full-Request   = Request-Line             ; Section 5.1
                 *( General-Header        ; Section 4.3
                  | Request-Header        ; Section 5.2
                  | Entity-Header )       ; Section 7.1
                 CRLF
                 [ Entity-Body ]          ; Section 7.2
```

```
Request-Header = Authorization            ; Section 10.2
               | From                     ; Section 10.8
               | If-Modified-Since        ; Section 10.9
               | Referer                  ; Section 10.13
               | User-Agent               ; Section 10.15
```

Exercise – HTTP 1.0 Client

- Let’s code it: GET

```
1  import socket
2
3  HOST = 'httpbin.org'
4  PORT = 80
5
6  client = socket.socket()
7  client.connect((HOST, PORT))
8
9  print("Request")
10
11 message = "GET /status/418 HTTP/1.0 \r\n"
12 message = message + "HOST: httpbin.org \r\n";
13 message = message + "CONNECTION: close \r\n";
14 message = message + "\r\n";
15
16 print(message);
17
18 client.sendall(bytes(message, 'utf-8'))
19 data = client.recv(4096)
20 print(data.decode("utf-8"))
21 client.close()
```

```
Request-Header = Authorization          ; Section 10.2
               | From                   ; Section 10.8
               | If-Modified-Since      ; Section 10.9
               | Referer                ; Section 10.13
               | User-Agent             ; Section 10.15
```

Disclaimer: There may be better ways to code but for the purpose of demo the code is clear.

Exercise – HTTP 1.0 Client

- Let’s code it: POST

```python
import socket

HOST = 'httpbin.org'
PORT = 80

client = socket.socket()
client.connect((HOST, PORT))

print("Request")

body = '{"key1":value1,"key2":value2}'

message = "POST /post HTTP/1.0 \r\n"
message = message + "Content-Type:application/json\r\n";
message = message + "Content-Length:" + str(len(body))+ "\r\n";
message = message + "\r\n";
message = message + body;

print(message + "\n");

client.sendall(bytes(message, 'utf-8'))
data = client.recv(4096)
print(data.decode("utf-8"))
client.close()
```

```
Request       = Request-Line              ; Section 5.1
                *(( general-header        ; Section 4.5
                 | request-header         ; Section 5.3
                 | entity-header ) CRLF)  ; Section 7.1
                CRLF
                [ message-body ]          ; Section 4.3
```

Disclaimer: There may be better ways to code but for the purpose of demo the code is clear.

Exercise – HTTP 1.0 Client

- Review if the received response follows the specification RFC 2616.

Exercise – FTP

```python
import socket

HOST = '192.168.1.61'
PORT = 21

client = socket.socket()
client.connect((HOST, PORT))

client_port_number = client.getsockname()[1]
client_ip_address = client.getsockname()[0]

print("The client IP address and port number: {} {}".format(client_ip_address,client_port_number))

print(client.recv(4098).decode("utf-8"))

message = "USER cristian\r\n"

print("\n" + message + "\n");

client.sendall(bytes(message, 'utf-8'))

print(client.recv(4098).decode("utf-8"))

message = "PASS hiddenpassword\r\n"

print("\n" + message + "\n");

client.sendall(bytes(message, 'utf-8'))

print(client.recv(4098).decode("utf-8"))

message = "TYPE I\r\n"

print("\n" + message + "\n");

client.sendall(bytes(message, 'utf-8'))

print(client.recv(4098).decode("utf-8"))
```

Disclaimer: There may be better ways to code but for the purpose of demo the code is clear.

Exercise – FTP

```
39
40  listen_port = 39851
41  receive_response_socket = socket.socket()
42  receive_response_socket.bind((client_ip_address,listen_port));
43
44  ftp_ip = str(client_ip_address).replace(".",",")
45  ftp_port = str(int("{0:x}".format(listen_port)[:2], 16)) + "," + str(int("{0:x}".format(listen_port)[2:],16))
46
47  message = "PORT " + ftp_ip + "," + ftp_port +"\r\n"
48
49  print("\n" + message + "\n");
50
51  client.sendall(bytes(message, 'utf-8'))
52
53  print(client.recv(4098).decode("utf-8"))
54
55  message = 'LIST\r\n'
56
57  print("\n" + message + "\n");
58
59  client.sendall(bytes(message, 'utf-8'))
60  receive_response_socket.listen()
61  print(receive_response_socket.accept()[0].recv(4098).decode("utf-8"))
62
63  receive_response_socket.close()
64
65  message = "QUIT\r\n"
66
67  print("\n" + message + "\n");
68
69  client.sendall(bytes(message, 'utf-8'))
70
71  print(client.recv(4098).decode("utf-8"))
72
73  client.close();
```

Disclaimer: There may be better ways to code but for the purpose of demo the code is clear.

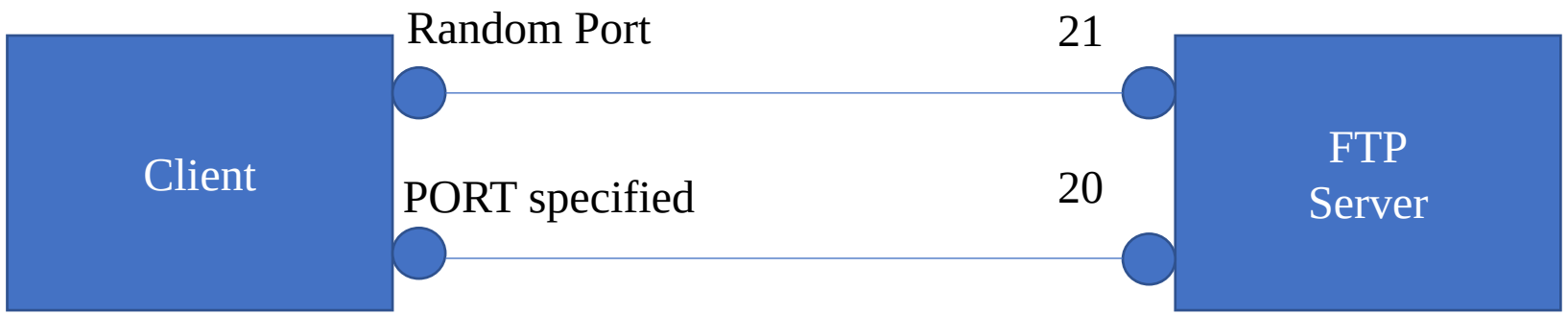
Exercise – FTP – PORT

- Let’s review RFC 959 to understand how the port must be sent.

```
DATA PORT (PORT)

   The argument is a HOST-PORT specification for the data port
   to be used in data connection.  There are defaults for both
   the user and server data ports, and under normal
   circumstances this command and its reply are not needed.  If
   this command is used, the argument is the concatenation of a
   32-bit internet host address and a 16-bit TCP port address.
   This address information is broken into 8-bit fields and the
   value of each field is transmitted as a decimal number (in
   character string representation).  The fields are separated
   by commas.  A port command would be:

      PORT h1,h2,h3,h4,p1,p2

   where h1 is the high order 8 bits of the internet host
   address.
```

Exercise – FTP – PORT

- Ports.



Exercise – FTP – PORT

- Let's review RFC 959 to understand how the port must be sent.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
1	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**1**

39851 = $2^{15} + 2^{12} + 2^{11} + 2^{9} + 2^{8} + 2^{7} + 2^{5} + 2^{3} + 2^{1} + 2^{0}$

7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1
1	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**1**

$2^{7} + 2^{4} + 2^{3} + 2^{1} + 2^{0} = 155$

$2^{7} + 2^{5} + 2^{3} + 2^{1} + 2^{0} = 171$

Recap

- Client-Server architecture
- IRC Client installation
- Socket creation - Client
- Socket creation – Server – Blocking
- HTTP 1.0 Client GET
- HTTP 1.0 Client POST
- FTP Client exercise