DISTRIBUTED SYSTEM DESIGN

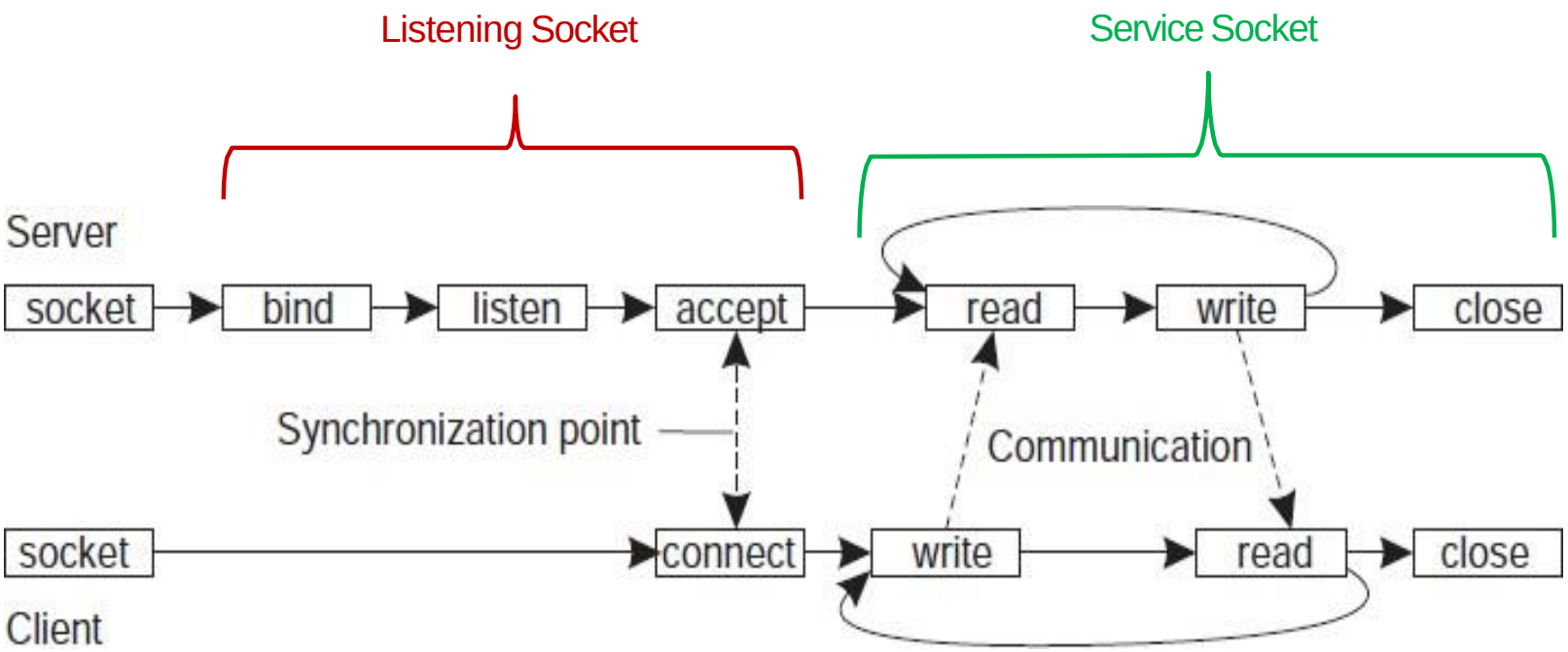
Lab 1

Socket Programming in Python I

Review: Communication via Sockets

- Sockets provide a communication mechanism between networked computers.
- A Socket is an end-point of communication that is identified by an IP address and port number.
- A client sends requests to a server using a client socket.
- A server receives clients' requests via a listening socket

Review: Communication via Sockets



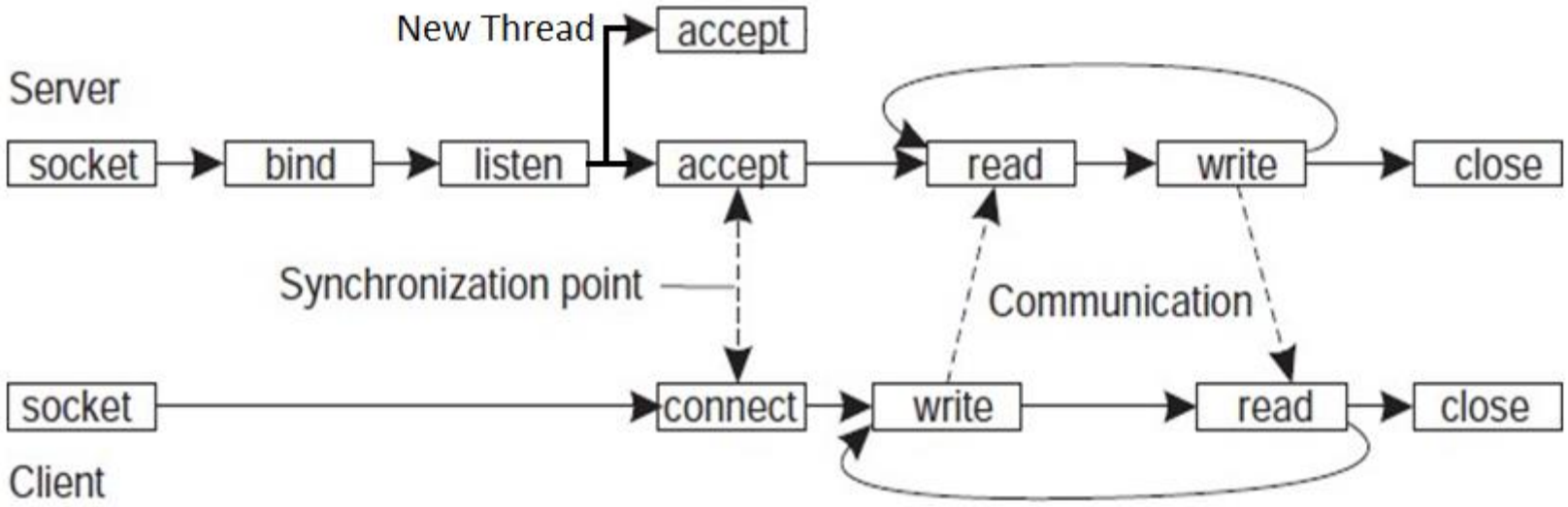
Review: Socket Methods

SN	Methods with Description
1	**socket.socket(*family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None*)** Create a new socket using the given address family, socket type and protocol number. The address family should be AF_INET (the default), AF_INET6, AF_UNIX, AF_CAN, AF_PACKET, or AF_RDS. The socket type should be SOCK_STREAM (the default), SOCK_DGRAM, SOCK_RAW or perhaps one of the other SOCK_ constants. The protocol number is usually zero and may be omitted or in the case where the address family is AF_CAN the protocol should be one of CAN_RAW, CAN_BCM, CAN_ISOTP or CAN_J1939.
2	**socket.create_server(*address, *, family=AF_INET, backlog=None, reuse_port=False, dualstack_ipv6=False*)** Convenience function which creates a TCP socket bound to *address* (a 2-tuple (host, port)) and return the socket object.
3	**socket.accept()** Accept 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.
4	**socket.bind(address)** Bind the socket to *address*. The socket must not already be bound.
5	**socket.connect(*address*)** Connect to a remote socket at *address*.
6	**socket.recv(*bufsize*[, *flags*])** Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by *bufsize*. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero.
7	***socket.sendall(bytes[, flags])*** Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for recv() above. Unlike send(), this method continues to send data from *bytes* until either all data has been sent or an error occurs. None is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.
8	**socket.close()** Mark the socket closed. The underlying system resource (e.g. a file descriptor) is also closed when all file objects from makefile() are closed. Once that happens, all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed).

Multi-Threaded Socket Applications

- Modern distributed systems perform multiple tasks in parallel.
- Servers:
  - Communicate with multiple clients.
  - Store/update records in a database.
  - Perform application logic.
- Clients:
  - Communicate with one or more servers.
  - Display UI to user.
- These systems leverage threaded programming to perform all tasks.
- Each task is carried out in a separate thread.

Multi-Threaded Socket in Servers



Threads in Python

- Threads can be created via the `Thread` class from the built-in `threading` library.

1. Create a class `MyCustomThread` that inherits the Thread class:
   a) Override the constructor (`__init__()`) to take the required arguments. A threaded socket operation should supply the client/service socket to the constructor.
   b) Override the `run()` method to perform the required task.
2. Create a `CustomThread` object and supply the required arguments.
3. Call the `start()` method from the object.
4. (optional) call `join()` method from the object in case the thread returns a value to ensure the thread has finished.

Thread Example

```
from threading import Thread

class MyThread(Thread):
    def __init__(self, message):
        Thread.__init__(self)
        self.message = message       # save the message argument
        self.return_val = None       # variable to store return value

    def run(self):
        user_input = input(self.message) # get input from the user (blocking)
        self.return_val = user_input     # store the input in return value

    def join(self, *args):
        Thread.join(self, *args)
        return self.return_val               # return the value upon join

myThread = MyThread('Enter user name.')
myThread.start()
print('Thread is now running.')
username = myThread.join()
print('Thread is done. Username is:', username)
```

Hello World Server

```
import socket

HOST = "127.0.0.1"  # localhost
PORT = 65432  # Port to listen on

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print(f"Connected by {addr}")
        while True:                         # handle all requests from the client
            data = conn.recv(1024)
            if not data:
                break
            message = f'I got "{data.decode()}" from you and I am sending it back.'
            conn.sendall(str.encode(message))
```

Threaded Hello World Server

```
import socket
from threading import Thread

class ClientThread(Thread):
    def __init__(self, service_socket : socket.socket,
                 address : str):
        Thread.__init__(self)
        self.service_socket = service_socket
        self.address = address

    def run(self):
        print(f"Connected by {self.address}")
        while True:  # handle all requests from the client
            data = self.service_socket.recv(1024)
            if not data:
                break
            message = f'I got "{data.decode()}" from you and I am sending it back.'
            self.service_socket.sendall(str.encode(message))
            print(f'Sent back "{data.decode()}" to: {self.address}')

        self.service_socket.close()

HOST = "127.0.0.1"  # localhost
PORT = 65432  # Port to listen on
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    while True:
        conn, addr = s.accept()
        client_thread = ClientThread(conn, addr)
        client_thread.start()
```

Hello World Client

```
import socket

HOST = "127.0.0.1"  # The server's hostname or IP address
PORT = 65432  # The port used by the server

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    print('Connected to:', HOST, PORT)
    message = input('Enter your message to server: ')    # user input to block the client
    s.sendall(message.encode())
    data = s.recv(1024)


print(f"Server says: {data.decode()}")
```

Exercises

1. Run the *Threaded Hello World* example.
2. Create a new threaded server `file_server.py` that takes a client message containing a file name (either “turtle.jpg” or “eagle.jpg”). It reads the corresponding file and sends it to the client as binary. It should listen on port 6565.
3. Modify the client as follows:
   a) Establishes a connection with the threaded hello world server.
   b) In a new thread:
      I. Takes user input: either “turtle.jpg” or “eagle.jpg”
      II. Opens a connection with `file_server.py`
      III. Gets the corresponding file as binary.
   c) Sends a message to the threaded hello world server: “The size of <FILE_NAME> is <FILE_SIZE_IN_BYTES>”, where the former is the name of the file and the latter is its size (length of the bytearray). You should get 55696 bytes for eagle.jpg and 583163 bytes for turtle.jpg