DISTRIBUTED SYSTEM DESIGN

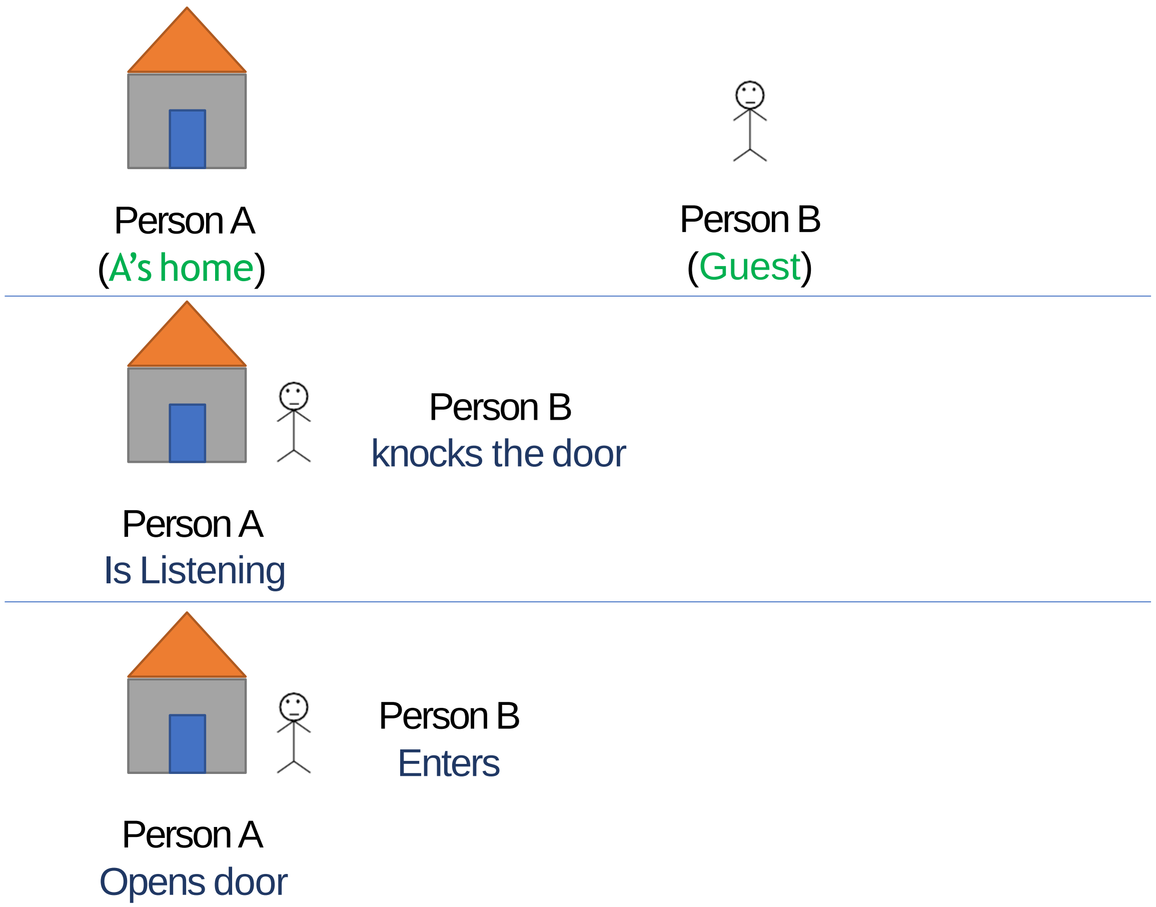
Lab 1

Socket Programming in Python I

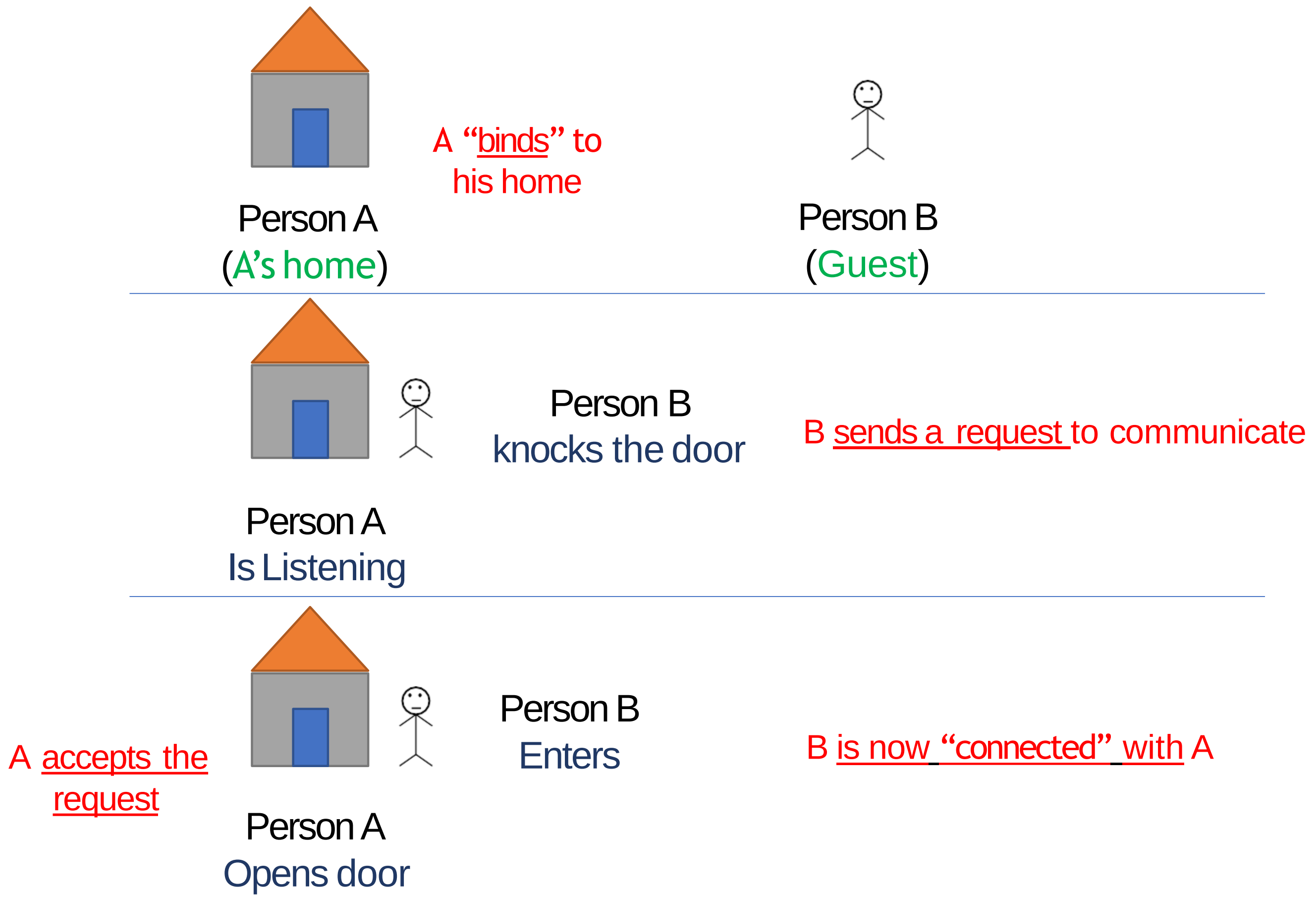
Communication via Sockets

- 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

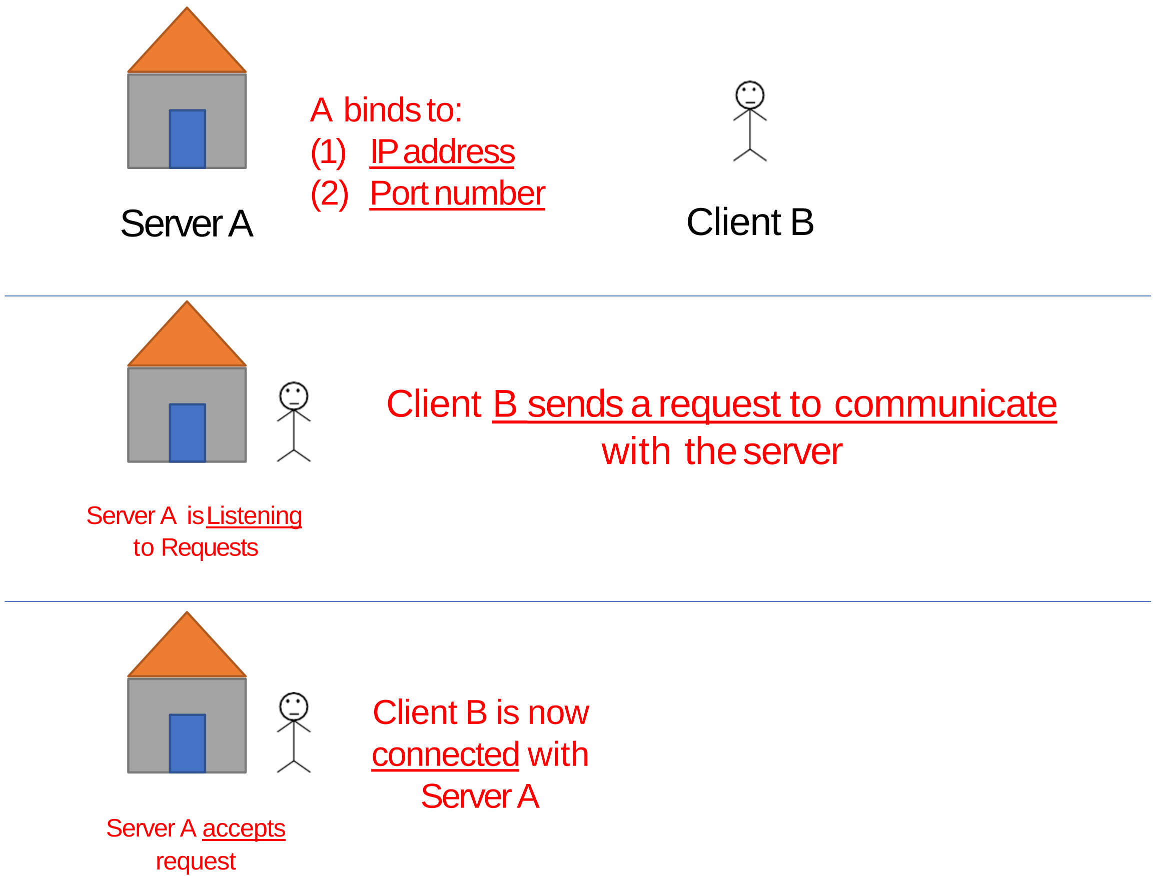
Communication via Sockets



Communication via Sockets



Communication via Sockets



Server A

A binds to:
(1) IP address
(2) Port number

Client B

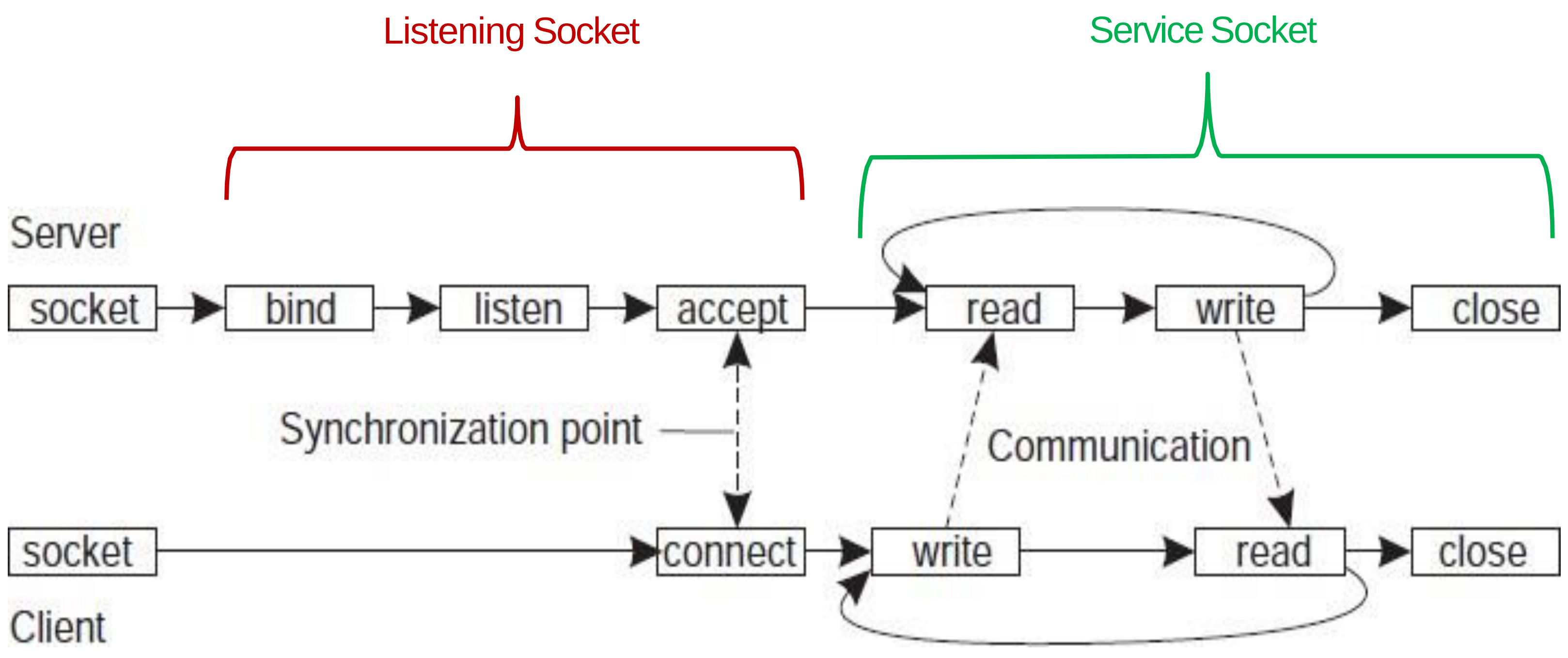
Server A is Listening to Requests

Client B sends a request to communicate with the server

Server A accepts request

Client B is now connected with Server A

Communication via Sockets



Socket Communication Recipe

1. Server instantiates a **`socket`** object. This socket is referred to as the listening socket.
2. Server *binds* its socket to a specific IP address and port.
3. Server invokes the **`accept()`** method that awaits incoming client connections.
4. Client instantiates **`socket`** object. This socket is referred to as a client socket.
5. Client *connects* to the server with IP address and port.
6. On the server side, **`accept()`** returns a new socket referred to as a service socket on which the client reads/writes.

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).

Transport Protocols

- Socket: endpoint to read and write data
- Each Socket has a network protocol
- Two types of protocols used for communicating data/*packets* over the internet:
  - TCP:
    - *Transmission Control Protocol*
    - Connection Oriented (*handshake*)
    - `socket.socket(type=SOCK_STREAM)`
  - UDP:
    - *User Datagram Protocol*
    - “Connectionless”
    - `socket.socket(type=SOCK_DGRAM)`

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))
```

Hello World Client

```python
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))
    s.sendall(b"Hello world")
    data = s.recv(1024)

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

Exercises

1. Run the *Hello World* example.
2. Modify the client to send one more message and receive the server’s response.
3. Modify the server to keep running after handling a client request.
4. Modify the client to send multiple numbers as a string (separated by spaces), the server calculates and returns their sum.
5. `socket.sendall()` ensures all the data is sent but `socket.recv(1024)` receives a maximum of 1024 bytes. Modify the client and server to handle messages of variable sizes. You might need to add an end-of-file token (e.g. `"<EOF>"`) to your messages to know when a message ends.