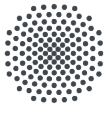


University of Stuttgart
Germany

Herman Hollerith Zentrum

Client-server architecture model

With sockets and multiprocessing in Python

Ilche Georgievski
ilche.georgievski@iaas.uni-stuttgart.de
Room: U38 0.353

2020
Summer

Architecture models

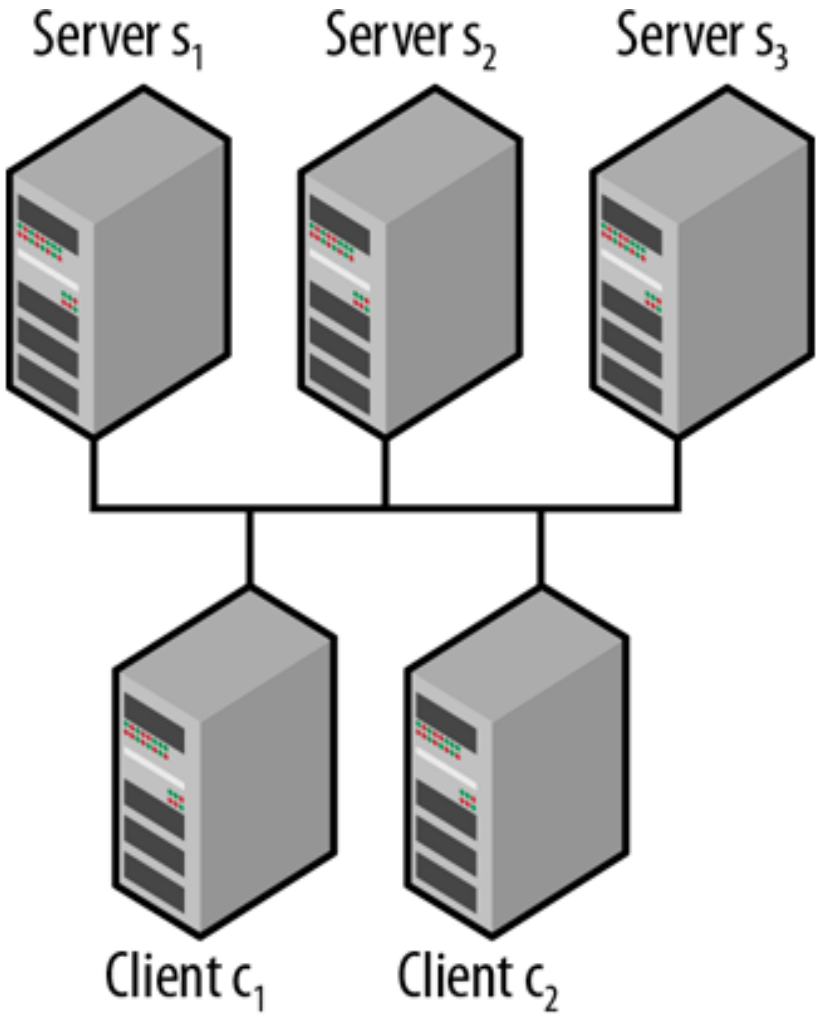
- Client-server architecture model
- Peer-to-peer architecture model

CLIENT-SERVER ARCHITECTURE MODEL

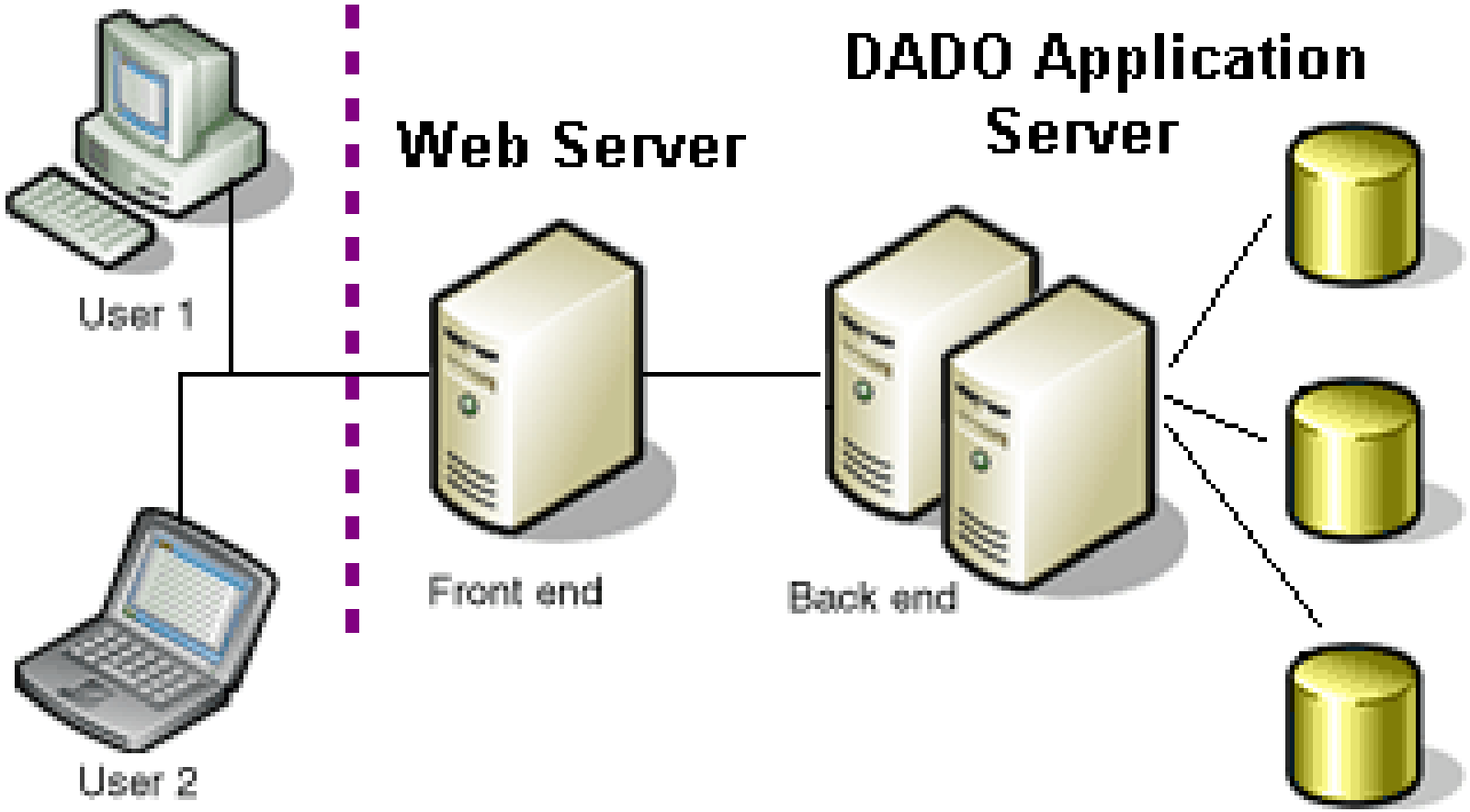
Client-server architecture model

- One server, one client
  - Toy model
  - Useful for explanations
  - Useless for real applications
- One server, multiple clients
  - Simplistic model
  - Useful for limited types of applications
- Multiple (and multiple types of) servers, multiple clients
  - Realistic model
  - Typical for distributed applications

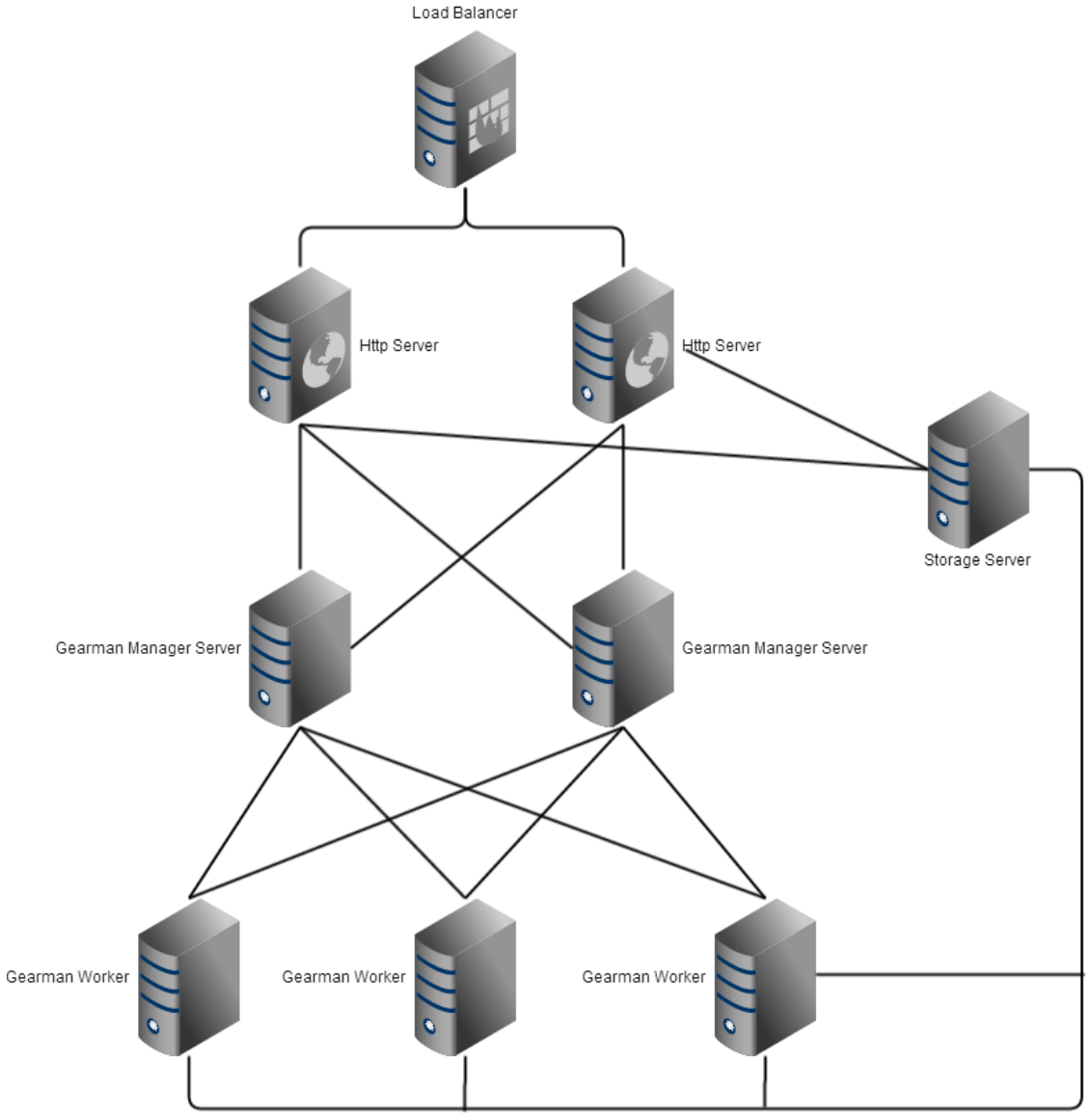
Client-server architecture model

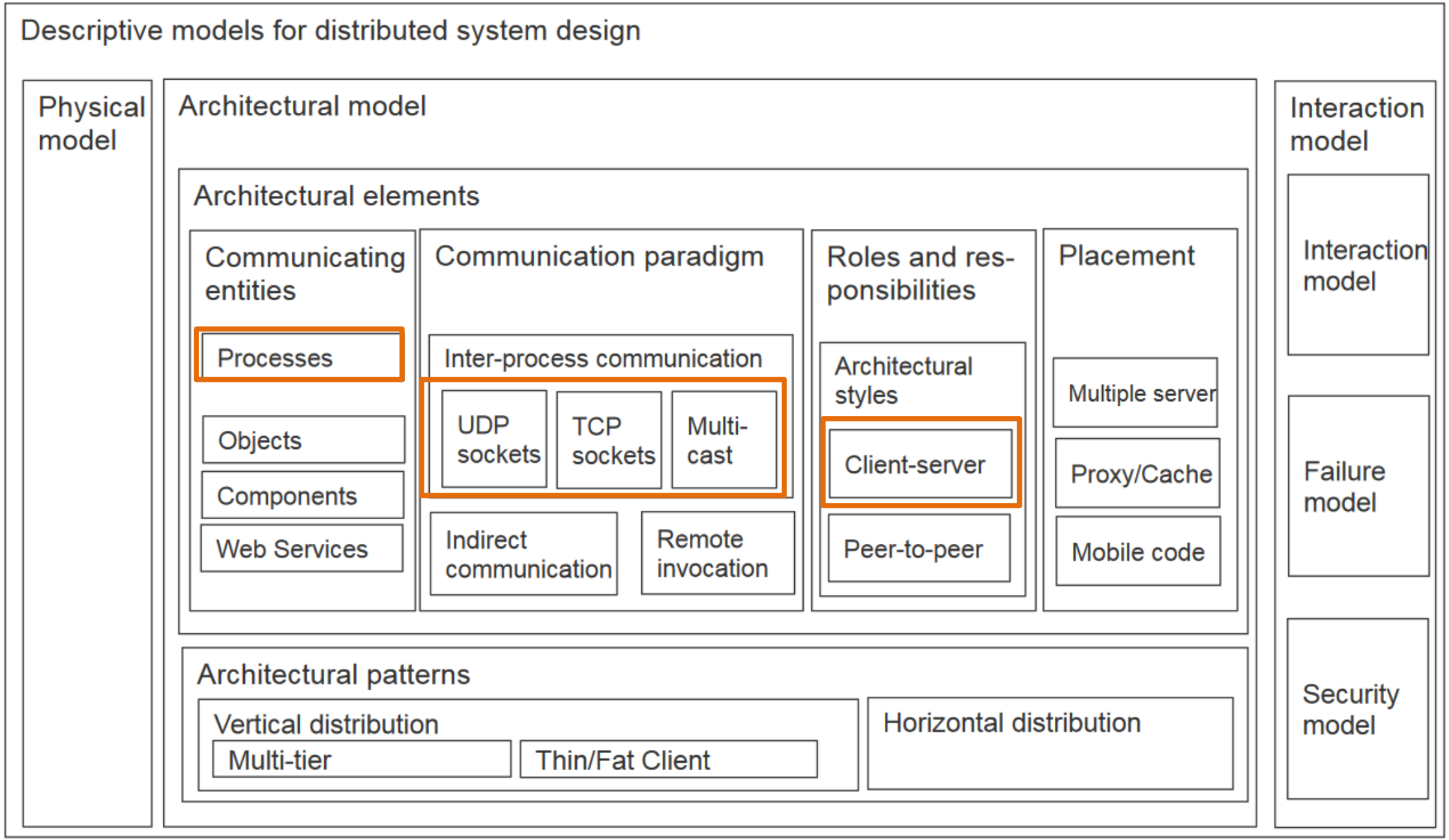


Client-server architecture model



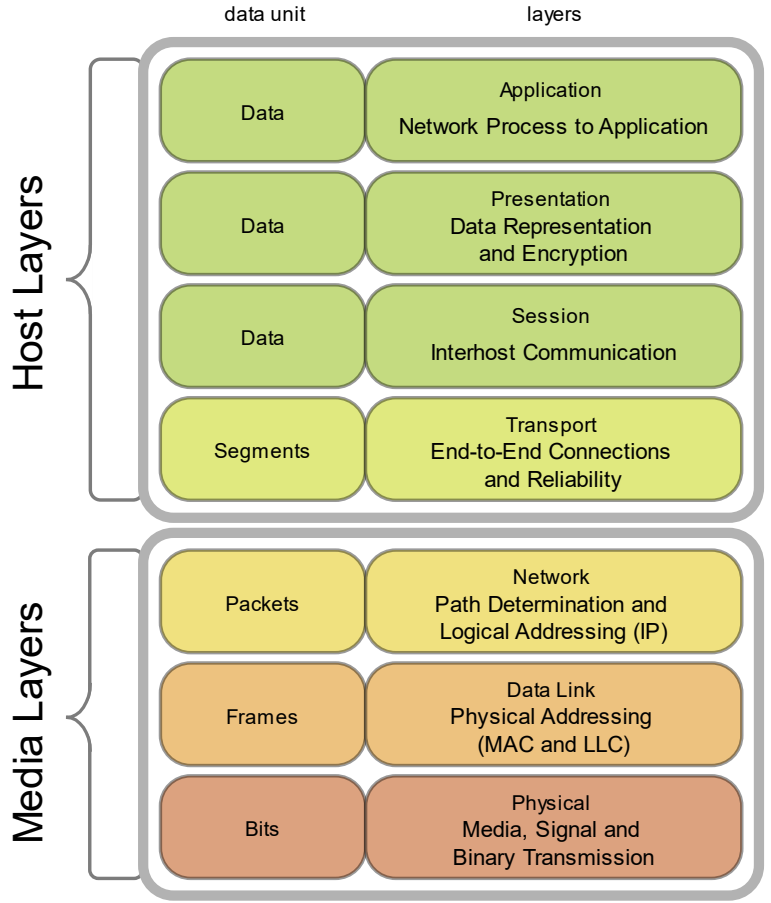
Client-server architecture model




Descriptive models for distributed system design
Physical model
Architectural model
Architectural elements
Communicating entities
Processes
Objects
Components
Web Services
Communication paradigm
Inter-process communication
UDP sockets
TCP sockets
Multi-cast
Indirect communication
Remote invocation
Roles and res-ponsibilities
Architectural styles
Client-server
Peer-to-peer
Placement
Multiple server
Proxy/Cache
Mobile code
Architectural patterns
Vertical distribution
Multi-tier
Thin/Fat Client
Horizontal distribution
Interaction model
Interaction model
Failure model
Security model

Applications

- Host
  - IP address of the machine
  - Network layer
- Port
  - Port number where the application runs
  - Transport layer



By Offnfopt - Own work, Public Domain,
https://commons.wikimedia.org/w/index.php?curid=39917431

Client-server architecture model

- **Server**
  - Application that passively waits for contact
  - Runs on big machines
  - Many clients simultaneously
- **Client**
  - Application that actively initiates contact
  - Runs on end-users‘ devices
- **Programming-level sockets**
  - Create and manage connections
  - *Send* and *receive* data through connections

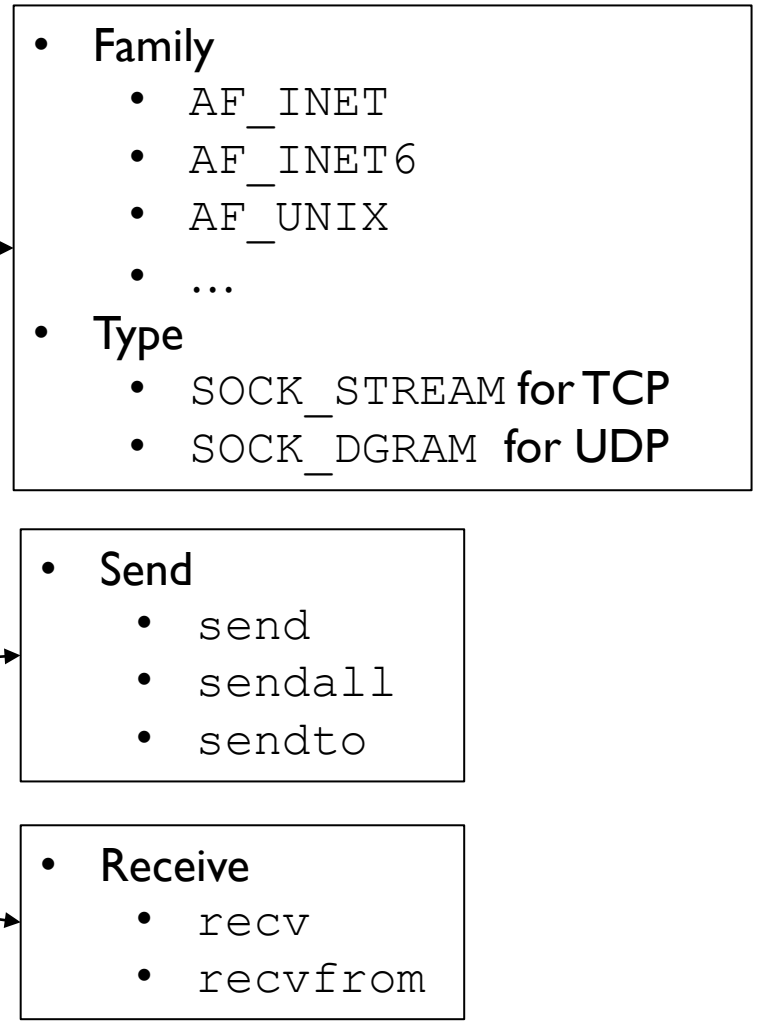
SOCKETS

Sockets

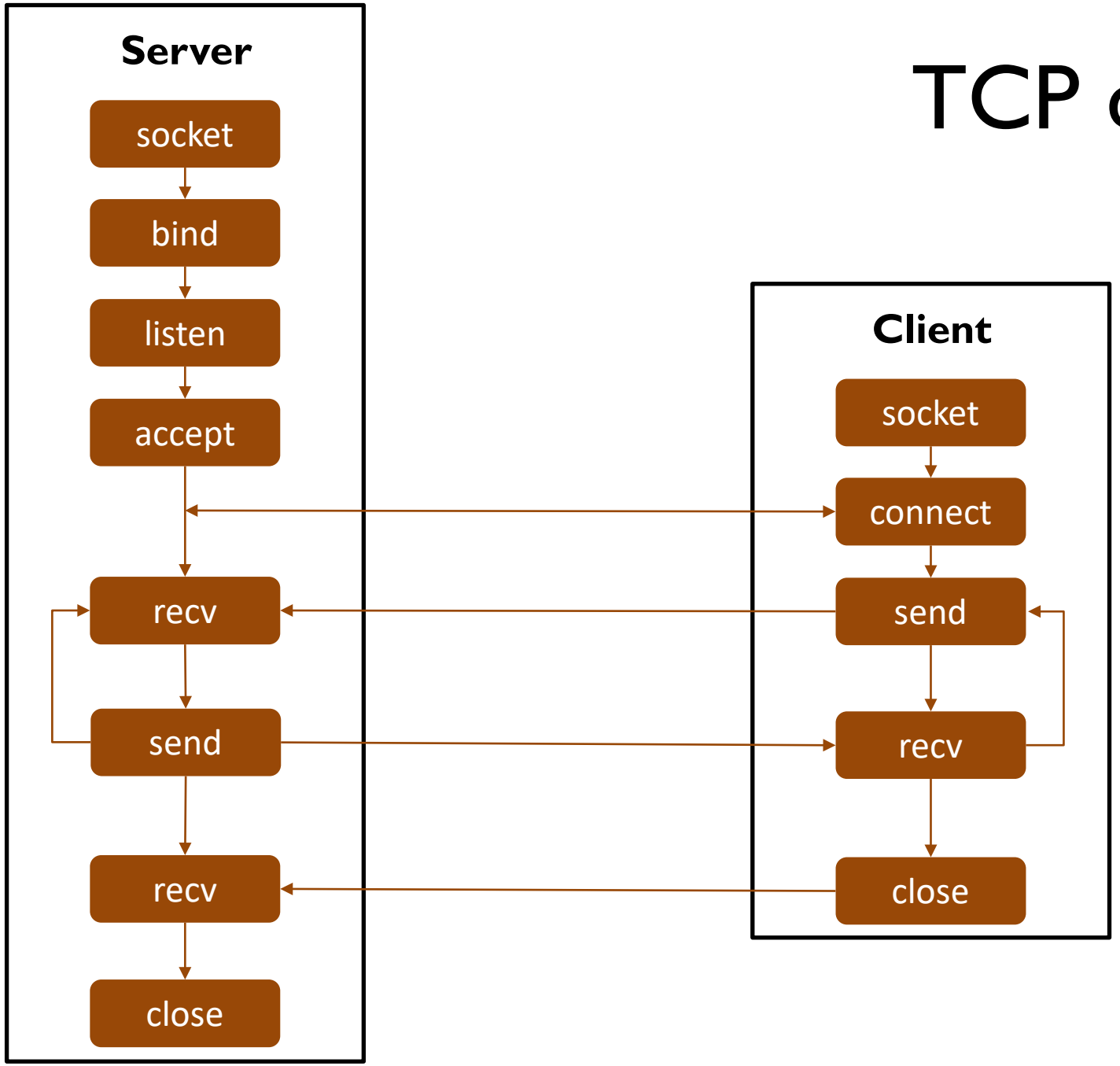
- OS-level data structure
  - Remote IP address
  - Remote port number
  - Local IP address
  - Local port number
  - Message queue

Sockets in Python

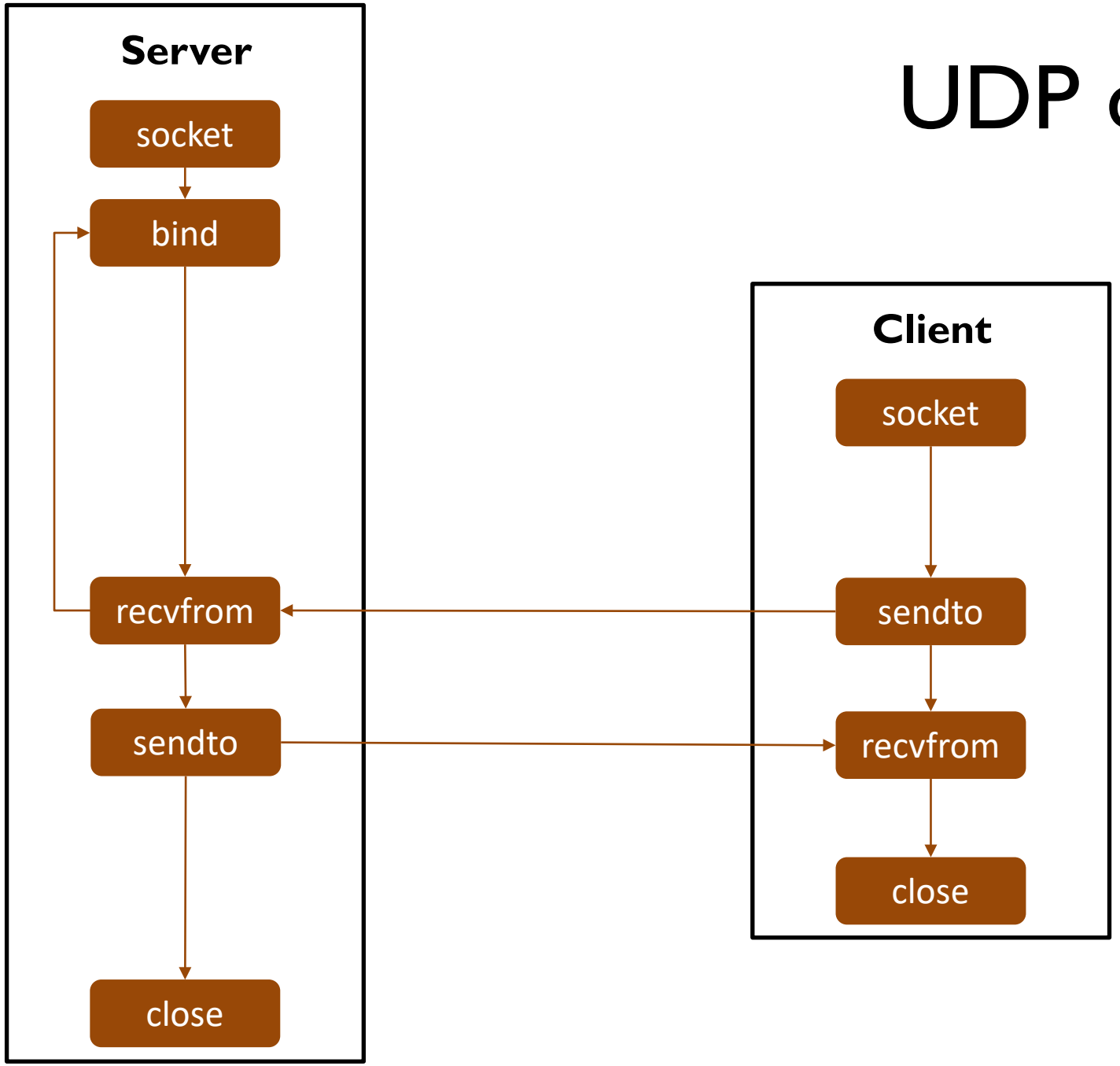
- TCP and UDP
- Basic socket API functions
  - `socket(family, type)`
  - `bind(address)`
  - `listen()`
  - `accept()`
  - `connect(address)`
  - `send(bytes)`
  - `recv(buffsize)`
  - `close()`



TCP data flow



UDP data flow



With sockets

ONE SERVER, ONE CLIENT

Server in Python

```
import socket

# Create a UDP socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind the socket to the port
server_address = '127.0.0.1'
server_port = 10001

# Buffer size
buffer_size = 1024

message = 'Hi client! Nice to connect with you!'


while True:
    print('\nWaiting to receive message...\n')

    data, address = server_socket.recvfrom(buffer_size)
    print('Received message from client: ', address)
    print('Message: ', data.decode())

    if data:
        server_socket.sendto(str.encode(message), address)
        print('Replied to client: ', message)
```

Client in Python

```python
import socket

# Create a UDP socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind the socket to the port
server_address = '127.0.0.1'
server_port = 10001

# Buffer size
buffer_size = 4096

message = 'Hi server!'

try:
    # Send data
    client_socket.sendto(message.encode(), (server_address, server_port))
    print('Sent to server: ', message)

    # Receive response
    print('Waiting for response...')
    data, server = client_socket.recvfrom(buffer_size)
    print('Received message from server: ', data.decode())

finally:
    client_socket.close()
    print('Socket closed')
```

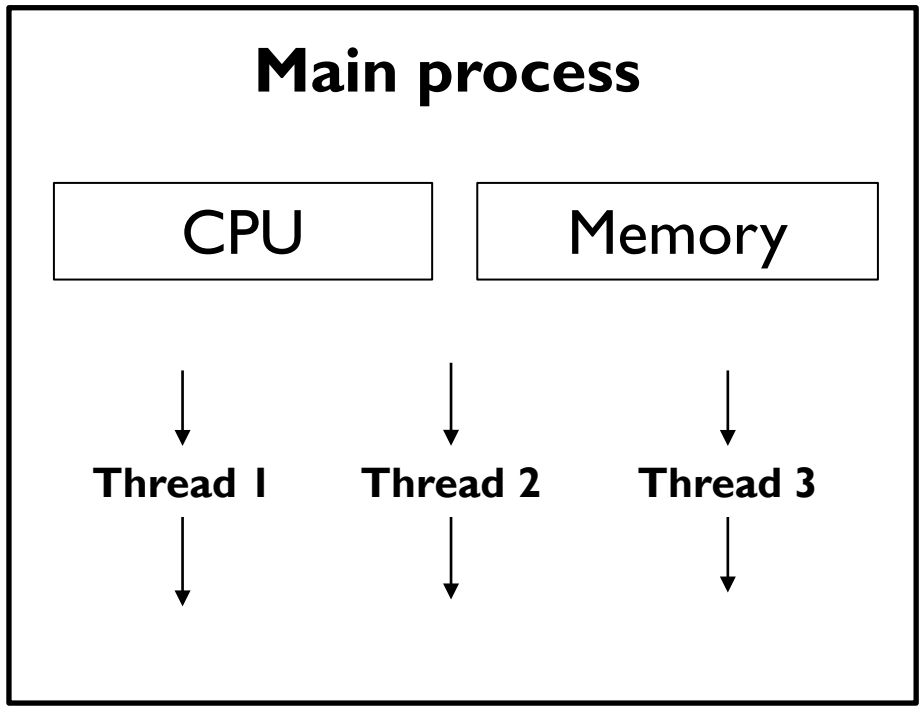
MULTIPROCESSING

Concurrency

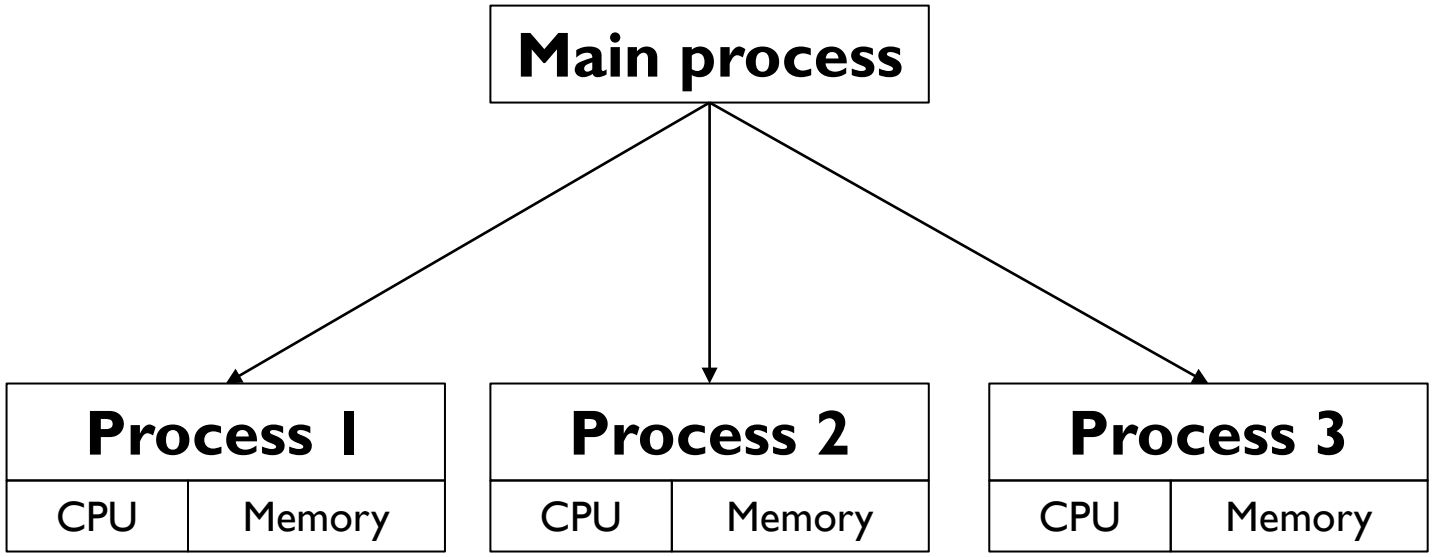
- **Program**
  - Source code for process(es)
- **Process**
  - Unit of program execution as seen by an OS
- **Thread**
  - Sequential flow of control within a process
  - Process can contain one or more threads
- **Multiprocessing**
  - Concurrent execution of several programs on one machine
- **Multithreading**
  - Execution of a program with multiple threads

Concurrency

Multithreading



Multiprocessing



Concurrency in Python

- Threads (`threading`)
- Processes (`multiprocessing`)
- Asynchronous I/O (`asyncio`)

Multiprocessing in Python

- Local and remote concurrency
- Enables spawning processes
  - `Process` class
- Process pool
  - `Pool` class
- Communication between processes
  - Queues
  - Pipes
- Synchronisation between processes
  - Standard synchronisation primitives (e.g., locks)

Process class

- `Process(target, name, args)`
  - `run()`
  - `start()`
  - `join()`

Processes in Python

```
from multiprocessing import Process
import os


def salute(course):
    print('Hello', course)
    print('Parent process id:', os.getppid())
    print('Process id:', os.getpid())


if __name__ == '__main__':
    p = Process(target=salute, args=('DS',))
    p.start()
    p.join()
```

With sockets

ONE SERVER, MULTIPLE CLIENTS

Exercise

Develop a client-server application that supports concurrency via multiprocessing. A client process sends a hello message to the server including its process ID. The server receives such a message and spawns a process that sends back a hello message with its own process ID.

Multiprocessing server

```
import multiprocessing
import socket
import os


class Server(multiprocessing.Process):
    def __init__(self, server_socket, received_data, client_address):
        super(Server, self).__init__()
        self.server_socket = server_socket
        self.received_data = received_data
        self.client_address = client_address

    def run(self):
        message = 'Hi ' + self.client_address[0] + ':' + str(self.client_address[1]) + '. This is server ' + str(os.getpid())
        self.server_socket.sendto(str.encode(message), self.client_address)
        print('Sent to client: ', message)


if __name__ == "__main__":
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Bind the socket to the port
    server_address = '127.0.0.1'
    server_port = 10001

    # Buffer size
    buffer_size = 1024

    print('Server up and running at {}:{}'.format(server_address, server_port))

    server_socket.bind((server_address, server_port))

    while True:
        data, address = server_socket.recvfrom(buffer_size)
        print('Received message \'{}\' at {}:{}'.format(data.decode(), address[0], address[1]))
        p = Server(server_socket, data, address)
        p.start()
        p.join()
```

Multiple client processes

```
import socket
import multiprocessing
import os


def send_message(server_address, server_port):
    try:
        client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

        message = 'Hi from ' + str(os.getpid()) + ' at ' + server_address + ':' + str(server_port)

        # Send data
        client_socket.sendto(str.encode(message), (server_address, server_port))
        print('Sent to server: ', message)

        # Receive response
        print('Waiting for response...')
        data, server = client_socket.recvfrom(1024)
        print('Received message: ', data.decode())

    finally:
        client_socket.close()
        print('Socket closed')


if __name__ == '__main__':

    # Bind the socket to the port
    server_address = '127.0.0.1'
    server_port = 10001

    for i in range(3):
        p = multiprocessing.Process(target=send_message, args=(server_address, server_port))
        p.start()
        p.join
```

Pool class

- Synchronous execution
  - `Pool.map()` and `Pool.starmap()`
  - `Pool.apply()`
- Asynchronous execution
  - `Pool.map_async()` and `Pool.starmap_async()`
  - `Pool.apply_async()`

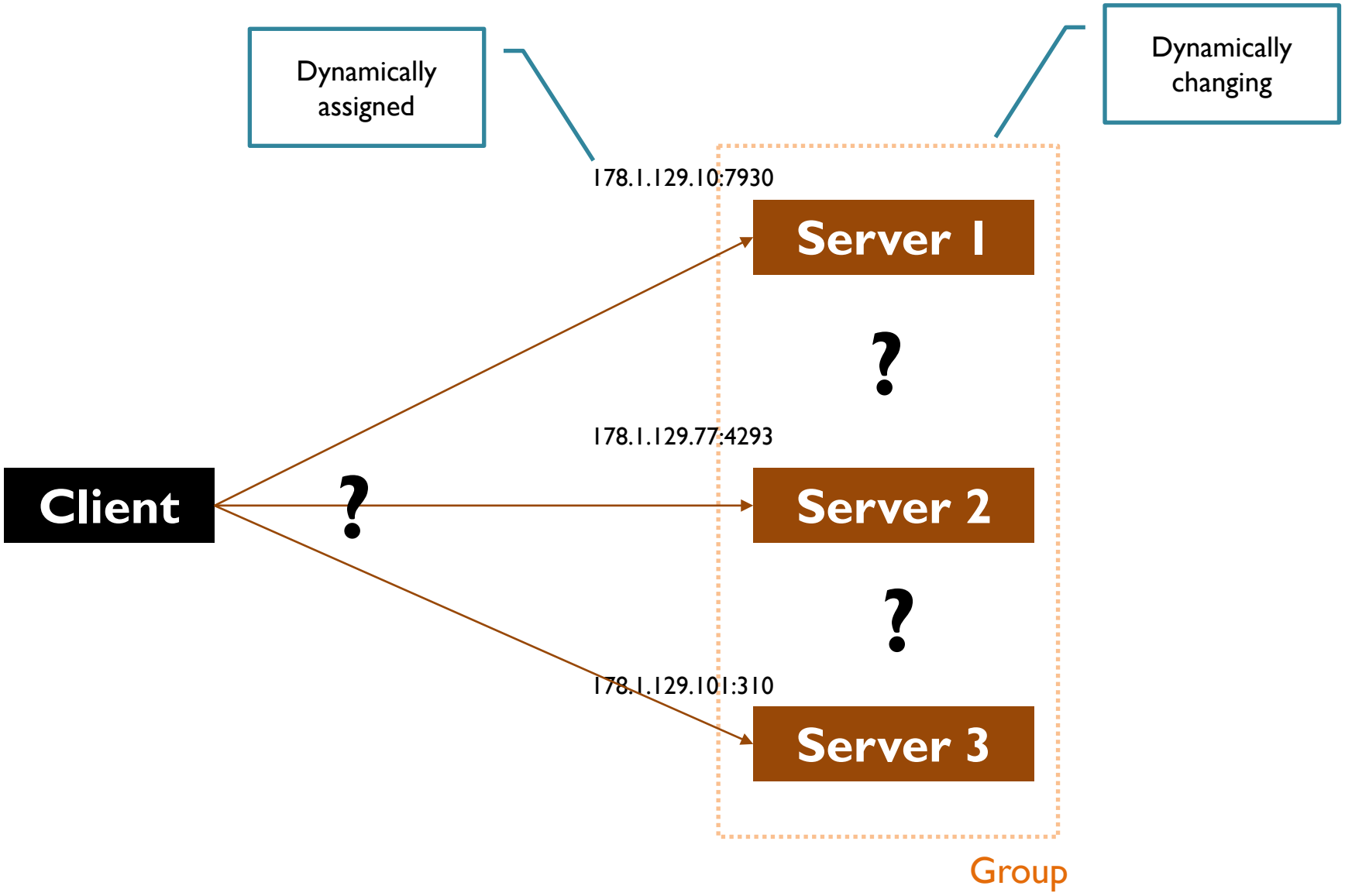
Asynchronous multiprocessing server

```python
import socket
import os
import multiprocessing


def send_message(server_socket, client_address):
    message = 'Hi ' + client_address[0] + ':' + str(client_address[1]) + '. This is server ' + str(
        os.getpid())
    server_socket.sendto(str.encode(message), client_address)
    print('Sent to client: ', message)


def test(x):
    return x * x


if __name__ == "__main__":

    # How many worker processes should be spawned
    number_processes = 4
    # Initialise the pool
    pool = multiprocessing.Pool(number_processes)

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    # server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    # Bind the socket to the port
    server_address = '127.0.0.1'
    server_port = 10001

    # Buffer size
    buffer_size = 1024

    print('Server up and running at {}:{}'.format(server_address, server_port))

    server_socket.bind((server_address, server_port))

    while True:
        data, address = server_socket.recvfrom(buffer_size)
        print('Received message \'{}\' at {}:{}'.format(data.decode(), address[0], address[1]))
        # `pool.apply` the `send_message()`
        pool.apply_async(send_message, args=(server_socket, address,))
```

Exercise

Develop a client-server application that supports asynchronous concurrency. The server should be able to receive requests from multiple clients at the same time. A client sends a number and receives back the square of that number.

MULTIPLE SERVERS, MULTIPLE CLIENTS


Dynamically assigned
Dynamically changing
178.1.129.10:7930
Server 1
?
178.1.129.77:4293
Client
?
Server 2
?
178.1.129.101:310
Server 3
Group

Considerations

- What kind of architecture model will be the system based on?
- What happens when a new server comes in?
  - How should a new server finds out about which are the existing servers in the group?
  - How should the existing servers find out about the new server?
- Which server to be contacted by a client?
  - Will there be a coordinator?
  - What will the responsibilities of the coordinator be?
  - How will be the coordinator chosen?

Considerations

- How should the system components communicate reliably?
  - What types of messages will be exchanged in the system?
  - What kind of communication reliability each type of messages requires?
  - Is causality among messages important?
- What happens when something goes wrong?
  - How to detect that a fault happened?
  - How will the system recover when a server crashes?
  - How will the system recover when the coordination crashes?
  - What if a client crashes?