LECTURE 9 SOCKET PROGRAMMING WITH PYTHON

Socket Programming in Python

- Socket programming is supported in the socket module in python
  - This is a wrapper around the OS socket API.
  - Python socket is easier to use than native OS socket.
- So, let's create our first socket.

```
from socket import *
s = socket(AF_INET, SOCK_STREAM) # or socket()
```

- Now we got an IPv4 TCP socket that you can use to connect to other machines.

Creating a socket end point: the socket function

- socket.socket([*family*, [*type*]]) creates a socket object. Remember, this is one endpoint of a two-way communication link.
- The family argument is the address family. The default is socket.AF_INET, which tells the socket to support the IPv4 protocol (32-bit IP addresses). Other choices include:
  - socket.AF_INET6 for IPv6 protocol (128-bit IP address).
- The type argument is the type of socket.
  - socket.SOCK_STREAM for reliable connection-oriented sockets (TCP).
  - socket.SOCK_DGRAM for unreliable datagram sockets (UDP).

Connecting to a remote end-point: the connect function

- After a socket is created, a network client program can use connect(*addr*) to connect to the socket end-point specified address addr. The addr argument is a tuple containing the host name/address (as a string) and port number (as an int).

```
s = socket()
s.connect(("websrv.cs.fsu.edu", 80))
```

Sending data: the send function

- send(bytes) returns the number of bytes sent.
- To send the whole string, Use sendall(bytes) blocks until all data has been transmitted. None is returned on success.
- Need to convert string into bytes using encode() or use b'…' for string literals.

```
s = socket()
s.connect(("websrv.cs.fsu.edu", 80))
s.send(b"GET /index.html HTTP/1.0\r\n\r\n")
```

Receiving data: the recv function

- recv(*bufsize*) receives and returns up to bufsize bytes of data from the connection.
  - Return empty string is the connection is closed.
  - Use decode() to convert bytes into string

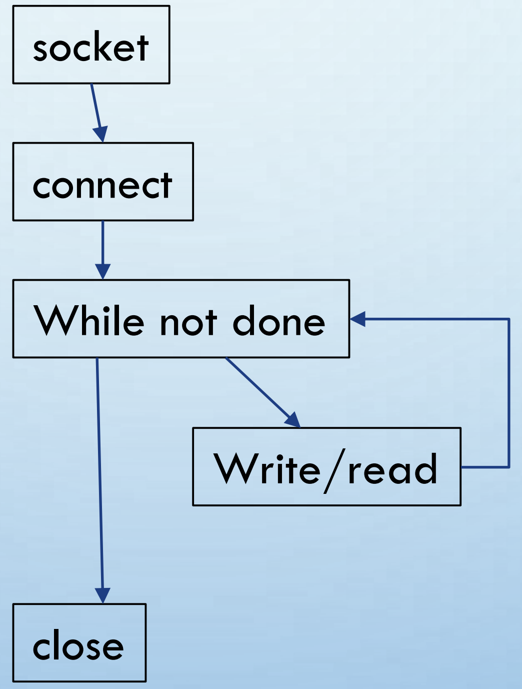
```
s = socket()
s.connect(("websrv.cs.fsu.edu", 80))
s.send(b"GET /index.html HTTP/1.0\r\n\r\n")
data = s.recv(100000)
print(data)
str = data.decode()
print(str)
```

Close socket: the close function

- Like a file, a socket need to be closed when it is done

```
s = socket()
s.connect(("websrv.cs.fsu.edu", 80))
s.send(b"GET /index.html HTTP/1.0\r\n\r\n")
data = s.recv(1000000)
print(data)
s.close()
```

The control flow of major socket calls in a network client



Binding a socket to address+port: the bind function

- bind(*addr*) binds the socket object to an address addr. As before, addr is a tuple containing hostname or host IP address and port.
  - Done by the network server program to obtain a well-known port
  - Optional for the network client
    - ❖ If not used, a system assigned random port is assigned to the socket when making connection.

```
from socket import *
s = socket()
h = socket.gethostname()
s.bind((h, 9000))

# We could also bind to localhost:
# s.bind(("localhost", 9000))
# s.bind(("127.0.0.1", 9000))
# Or bind to any address the machine has:
# s.bind(("", 9000))
```

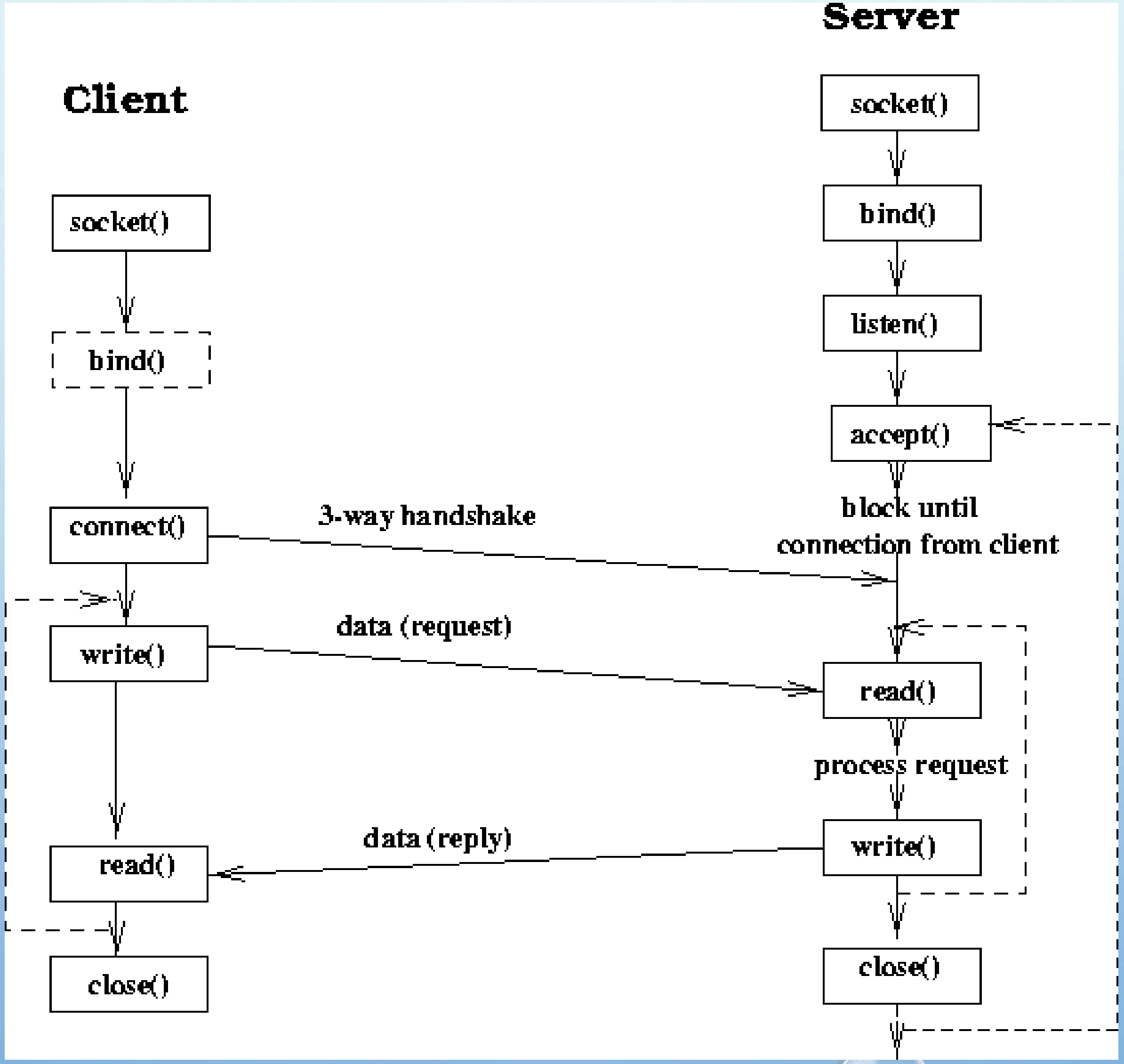
Turn a socket passive: the listen function

- There two are types of sockets, active socket and passive socket.
  - **Active socket**: sockets used to actively connect to another application and to send/recv data.
  - **Passive socket**: Sockets used by the server to accept connections from client. The passive socket only deals with connection, but not data communication. When a connection is accepted, a new active socket will be created at the server side for sending/receiving data.
- listen(*backlog*) makes the socket passive (listening for connections).
  - The backlog argument specifies how many connections to queue before connections become refused in case that the connections cannot be processed fast enough.

Accept a connection: the accept function

- accept() blocks the server program until a connection is made from a client and accepts the connection, then returns.
  - The return value is a pair(conn, address) where conn is a new active 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 (client's IP address and port number).

Socket functions in Client and Server



Analogy between socket and phone calls

Client

Socket calls	Phone analogy
socket	Pickup the phone
connect	Dial the number
write/read	Talk/listen
close	hang up

Server

Socket calls	Phone analogy
socket	Install a phone
bind	Get a phone number
listen	Turn on the ringer, ready to receive phone calls
accept	Answer a call
write/read	Talk/listen
close	hangup

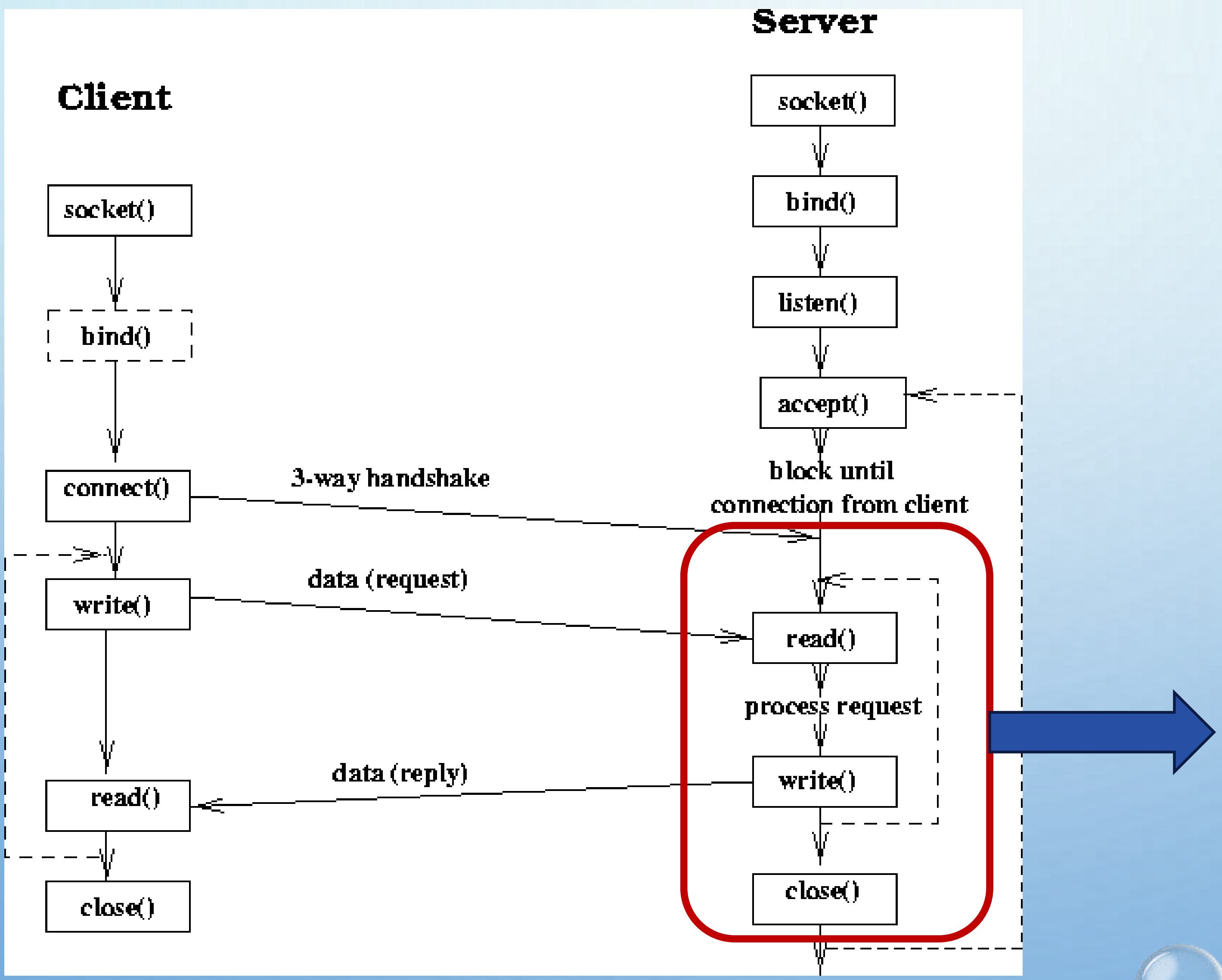
Echo Client and Server example

- See lect9/echo_server.py and lect9/echo_client.py

Concurrent Server

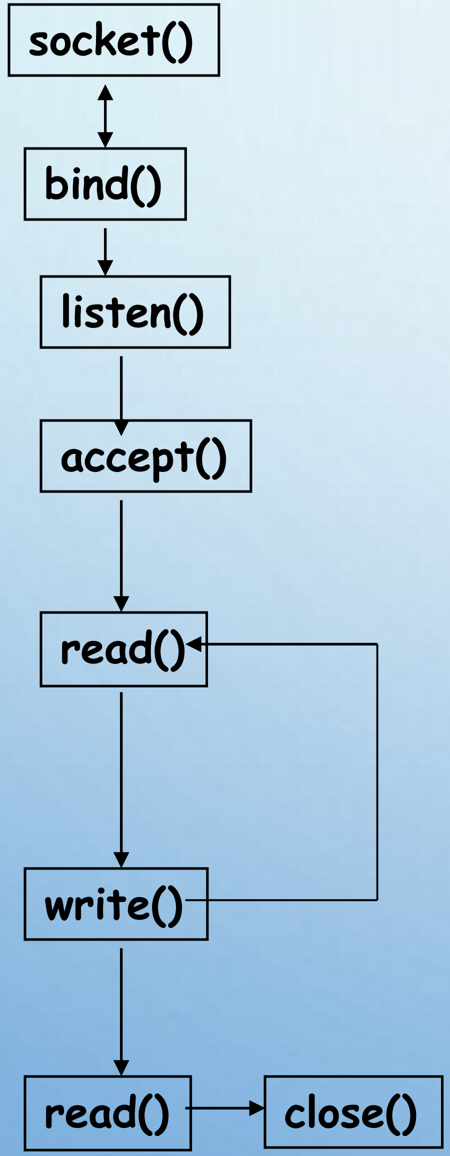
- A typical network server handles many connections simultaneously. But sequential server only handle one connection at a time.
- Extending the sequential server with multiprocessing is a simple way to enable the server to handle many connections
- A concurrent server starts a new child process/thread to handle a connection after it is accepted.
  - Advantages
    - simple program, most of the servers are implemented this way.
    - Almost no limits on the number of connection.
  - Main limitations: overheads, no shared state among child processes

Extend a sequential server to a concurrent server



Put this block in a process

Sequential server



socket()

bind()

listen()

accept()

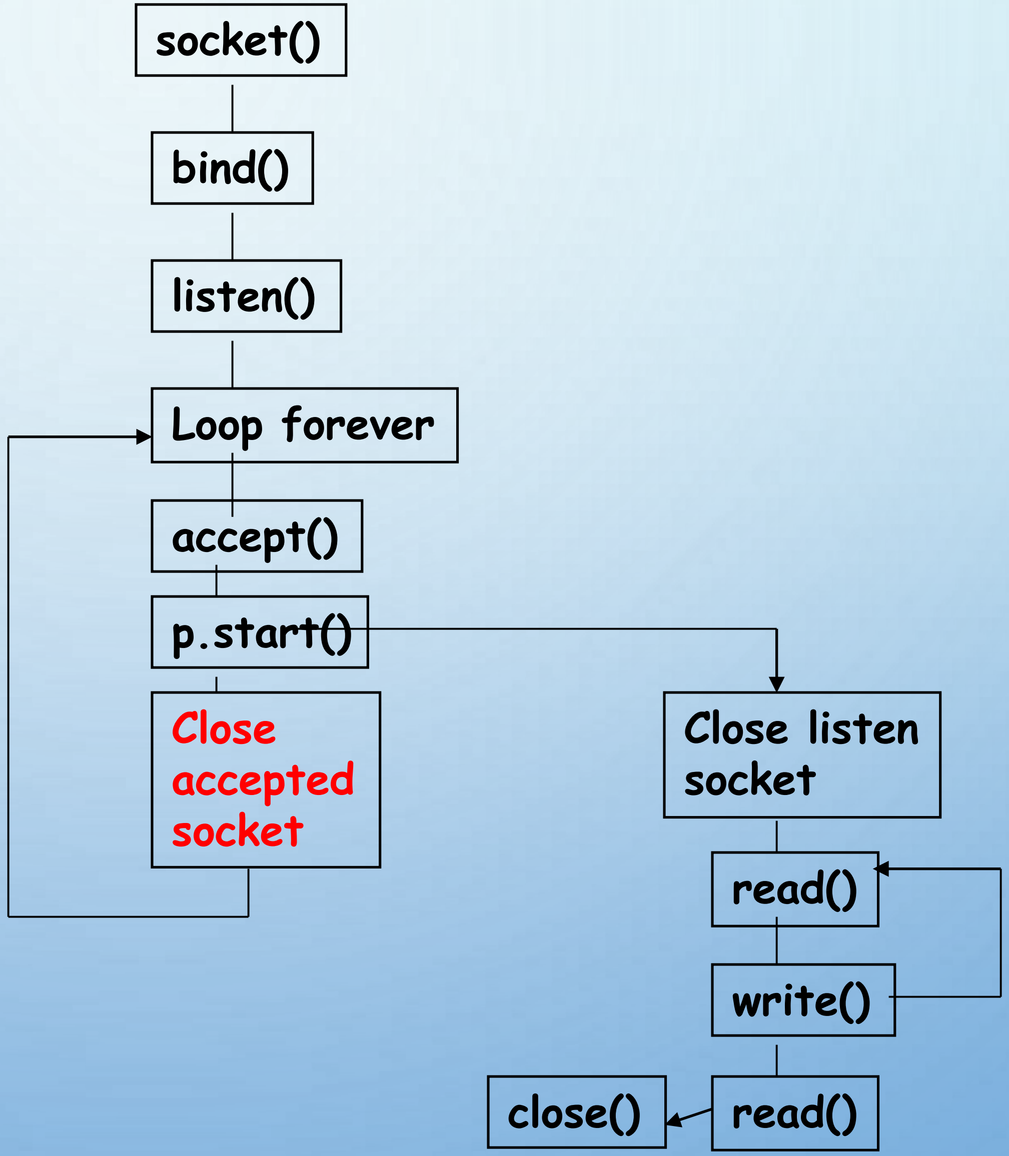
read()

write()

read()

close()

Concurrent server (mp_echo_server.py)



socket()

bind()

listen()

Loop forever

accept()

p.start()

Close accepted socket

Close listen socket

read()

write()

read()

close()

Use threads to implement current server

- Overcome some limitations of multiprocessing
- See mt_echo_server.py
- All threads can share data structure: good for applications that require shared data such as a game server.