VirginiaTech
Invent the Future

Client-Server Model
Sockets

ECE 4564 - Network Application Design

Dr. William O. Plymale

ece
BRADLEY DEPARTMENT
of ELECTRICAL & COMPUTER ENGINEERING

Office Hours

Monday – 11:00am-1:00pm - 360 Durham Hall

GTAs
Wednesday – 1:30pm-3:30pm
Thursday – 1:00pm–3:00pm
222 Whittemore Hall

Quiz Schedule

Quiz 1 - Wednesday (09/14)
Quiz 2 – Wednesday (10/05)
Quiz 3 – Wednesday (10/26)
Quiz 4 – Wednesday (11/16)

- In-class
- Canvas assessment
- Open notes (lecture slides on Canvas)
- Multiple-choice, True/False
- Requires writing and running Python code
  - Bring your laptops (fully-charged)
  - Network access (wireless)

Quiz 1

Quiz will cover following material

- Ubiquitous/Pervasive/Physical Computing
- Internet of Things
- Networks and Services
  - Protocols
  - TCP/IP and OSI Reference Model
- Client-Server Model
- Sockets
- Raspberry Pi / GPIO

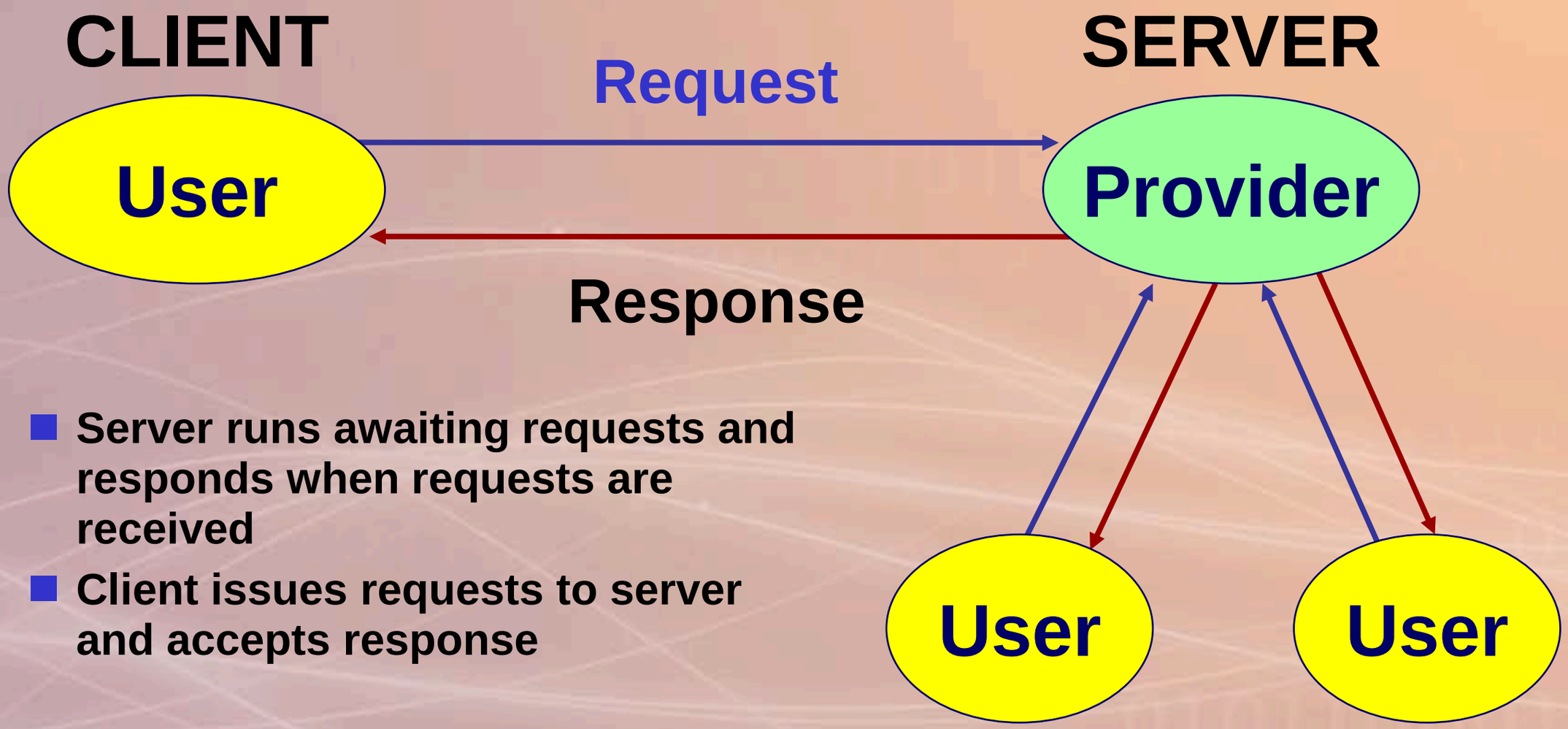
Previously

- Layered Network Models

Today

- Client-Server Model
- Sockets

Service User Versus Service Provider

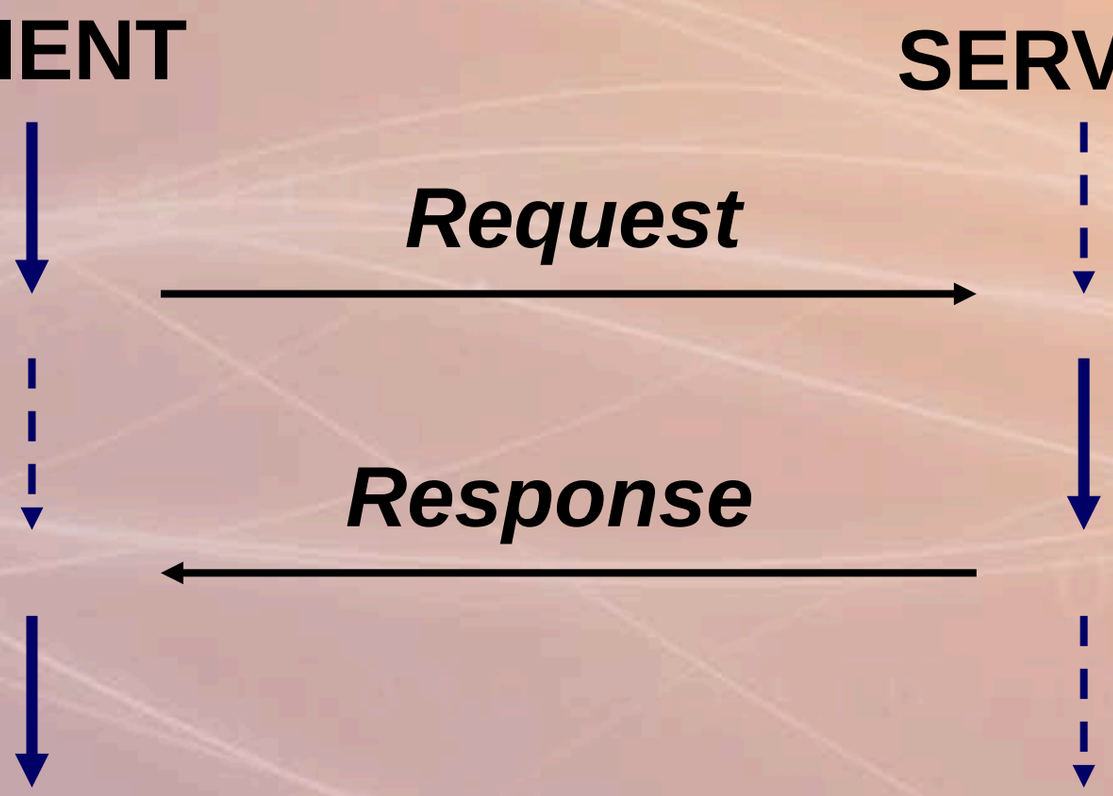


- Server runs awaiting requests and responds when requests are received
- Client issues requests to server and accepts response

There may be multiple users of one provider

Client-Server Model

- Client initiates peer-to-peer communication (at TCP- or UDP-level)
- Server waits for incoming request



Clients Versus Servers (1)

- Servers
  - Process started on a computer system.
  - Initializes itself, then goes to sleep.
  - Waits for client process to contact it requesting some service.
- Clients
  - Process started on same system, or a different system connected to the server's system by a network.
  - Often initiated by an interactive user.
  - Client process sends a request to the server process requesting some service.

Clients Versus Servers (2)

- Clients
  - Relatively simple (with respect to network communication)
  - User-level programs that require no special privileges
- Servers
  - More complex than clients due to performance and security requirements
  - Often require special system privileges
  - May run all the time or be started on-demand by operating system mechanisms, e.g. inetd in UNIX

Server Processes

- Iterative
  - Client request can be handled by the server in a known, short amount of time.
  - Server handles such a request itself.
  - Example: time-of-day service.

- Concurrent
  - Amount of time to service a request depends on the request itself.
  - Server does not know how much effort is required to honor request ahead of time.
  - Server invokes another process to handle each client request.
  - Original server process returns to sleep and listen.
  - Example: File operation.

Examples of a Service

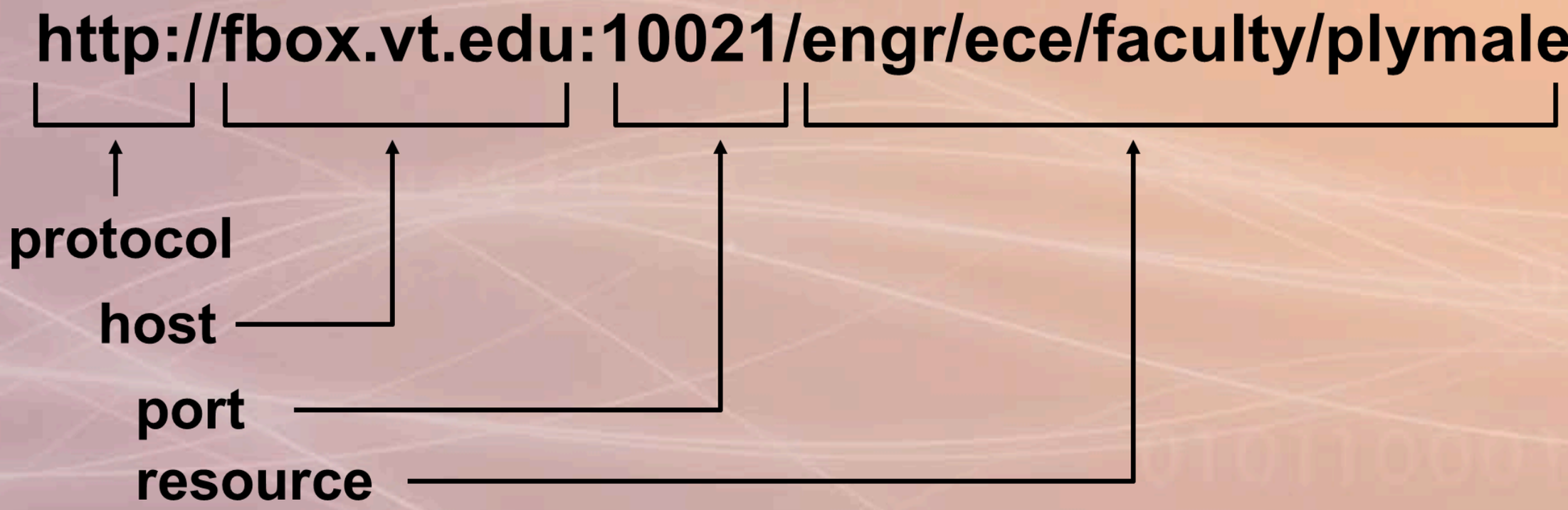
- Print a file on a printer for the client.
- Execute a command for the client on the server's system.
- Return the time-of-day to the client.
- Read or write a file on the server's system for the client.
- Allow the client to login to the server's system.

Client Parameterization

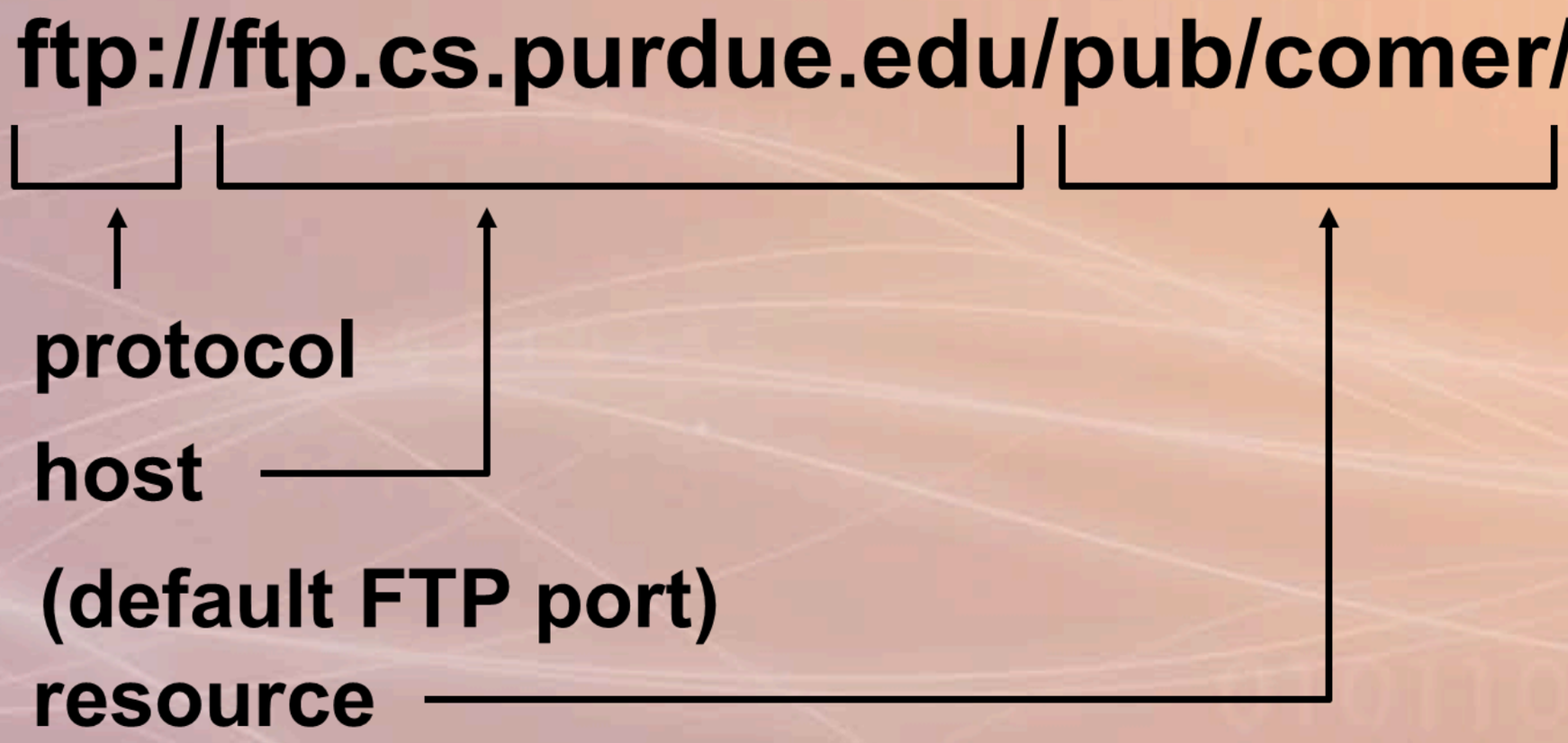
- Parameterized clients lead to generality, e.g. as in TELNET client being able to access other services
- Parameters
  - Destination host
    - Host name: fiddle.visc.vt.edu
    - IP address: 128.173.92.30
  - Port number (not just default)
  - Protocol- or application-specific information, e.g. block size

Universal Resource Locators (1)

URLs integrate many parameters



Universal Resource Locators (2)



Connection/Connectionless-oriented (1)

- *Connection-oriented* servers
  - Client must first connect to the server prior to any data transfer
  - Based on TCP (usually) -- reliable, but at the expense of connection overhead
    - Data arrives correctly
    - Data ordering is maintained
    - Data is not duplicated

Connection/Connectionless-oriented (2)

- *Connectionless* servers
  - Data can be sent by clients immediately
  - Based on UDP (usually) -- no connection overhead, but no benefits
    - Data may not arrive (unlikely in a LAN)
    - Data may be incorrect
    - Duplicates may arrive
    - May arrive out of order

Stateless / Stateful

- State information is any information about ongoing interactions
- *Stateful servers* maintain state information
- *Stateless servers* keep no state information
- Examples -- stateful or stateless?
  - HTTP?
  - FTP?

Stateless / Stateful

Stateless

```
//The state is derived by what is passed into the function
function int addOne(int number)
{
    return number + 1;
}
```

Stateful

```
//The state is maintained by the function
private int _number = 0; //initially zero

function int addOne()
{
    _number++;
    return _number;
}
```

File Server Example

Consider a file server that supports four operations

- OPEN -- identify file and operation, e.g. read or write
- READ -- identify file, location in file, number of bytes to read
- WRITE -- identify file, location in file, number of bytes, data to write
- CLOSE -- identify file

File Server Example: Stateless

- Stateless version -- identify all information with each request
- Example
  - OPEN(/tmp/test.txt, “r”)
  - READ(/tmp/test.txt, 0, 200)
  - READ(/tmp/test.txt, 200, 200)
- Redundant information is provided with subsequent requests
  - Inefficient with respect to information transfer
  - Server operation is simplified

File Server Example: Stateful (1)

- Stateful version -- server provides *handle* to access state at the server
- File open
  - Request: OPEN(/tmp/test.txt, “r”)
  - Reply: OPEN(ok, 32) -- handle = 32
  - State: 32: /tmp/test.txt, 0, read
- File read
  - Request: READ(32, 200)
  - Reply: READ(ok, data)
  - State: 32: /tmp/test.txt, 200, read

File Server Example: Stateful (2)

- What if there is a duplicate request?
  - READ(32, 200) sent once, but received twice
  - Client and server lose synchronization -- server thinks that 400 bytes have been read, client thinks it has read just 200 bytes
- Stateful servers are more complex than stateless servers since they must deal with synchronization
- State is implied by the protocol, not the implementation
  - TCP is a stateful protocol
  - Synchronization required with byte numbers

Reliable Operation

- Requirements for reliable operation
  - Every message received must be unambiguous
  - Operations cannot depend on the delivery or non-delivery of previous messages
  - Operation cannot depend on the order of delivery

- *Idempotent operation*: message gives the same result no matter how many times it arrives
  - READ(/tmp/test.txt, 0, 200)

Idempotent vs NonIdempotent

- Idempotent Operation
  - Return the time-of-day.
  - Calculate the square root of a number.
  - Return the current balance of a bank account.

- *Nonidempotent Operation*:
  - Append 512 bytes to the end of a file.
  - Deduct an amount from a bank account.

Stateful Protocol Design Issues

- Time-outs
- Duplicate requests and replies
- System crashes (at one end)
- Multiple clients
- File locking

Concurrency in Network Applications

- *Concurrency* is real or apparent simultaneous computing
  - Real in a multiprocessor
  - Apparent in a time-shared uniprocessor (apparent concurrency provided by OS)

- Networks are inherently concurrent --multiple hosts have the appearance of simultaneously transferring data

Client Concurrency

- Clients usually make use of concurrency in a trivial way
  - Multiple clients can run on a single processor
- Such concurrency is provided by the operating system, not by any programmed features of the client
- Note that complex clients can use concurrency, e.g. modern Web browser
  - Simultaneous requests and receipt of multiple files
  - Overlapping communication with graphical rendering or other processing

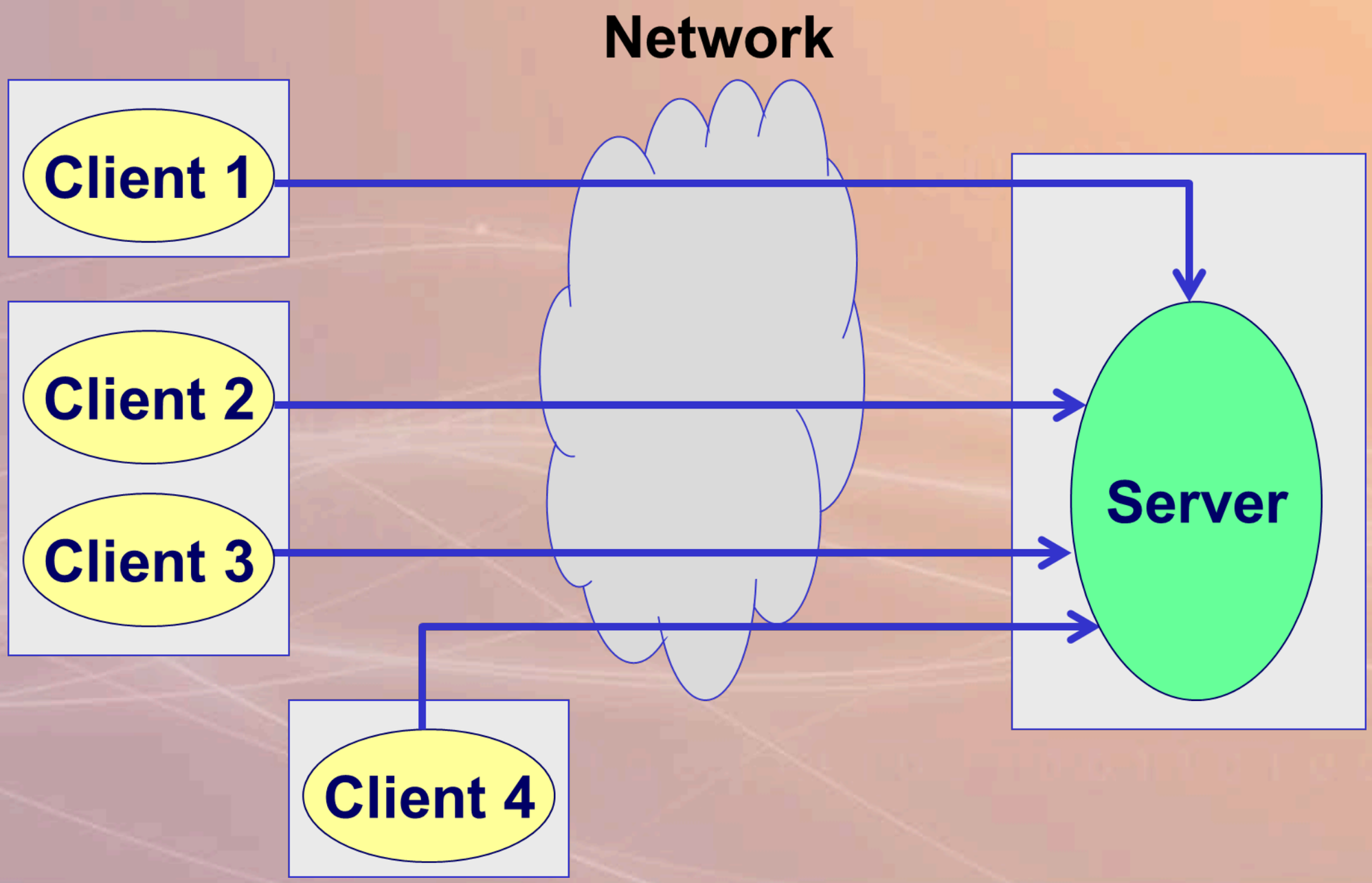
Concurrency at the Server

- Many servers provide concurrent operation
  - Apparent concurrency using asynchronous socket I/O
  - True (program-level) concurrency using multithreaded design

- Concurrency adds complexity!

- When is concurrency justified?
  - Need to simultaneously handle multiple requests
  - Need to respond to user input and/or output
  - Need to do other works, e.g., to delegate tasks or do other information processing

Server Concurrency (1)



Server Concurrency (2)

- Servers use concurrency to achieve functionality and performance
- Concurrency is inherent in the server -- must be explicitly considered in server design
- Exact design and mechanisms depend on support provided by the underlying operating system
- Achieved through
  - Concurrent processes
  - Concurrent threads

VirginiaTech
Invent the Future

Sockets

Three References

"Beej's Guide to Network Programming Using Internet Sockets" (PDF)

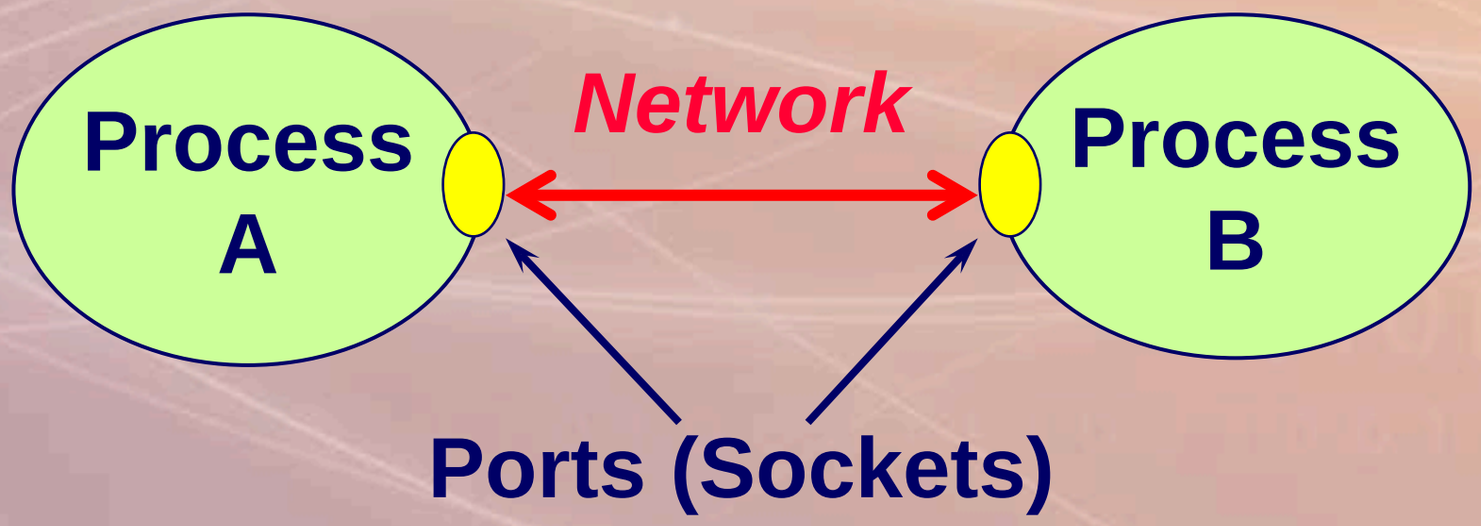
"Foundations of Python Network Programming", 3nd Ed. (Full text – VT Library)

Python Network Programming
(http://ilab.cs.byu.edu/python/)

Post other refs to Discussions

Socket Abstraction

- The *socket* is the basic abstraction for network communication in the socket API
  - Defines an endpoint of communication for a process
  - Operating system maintains information about the socket and its connection
  - Application references the socket for sends, receives, etc.



What Defines an Application Protocol

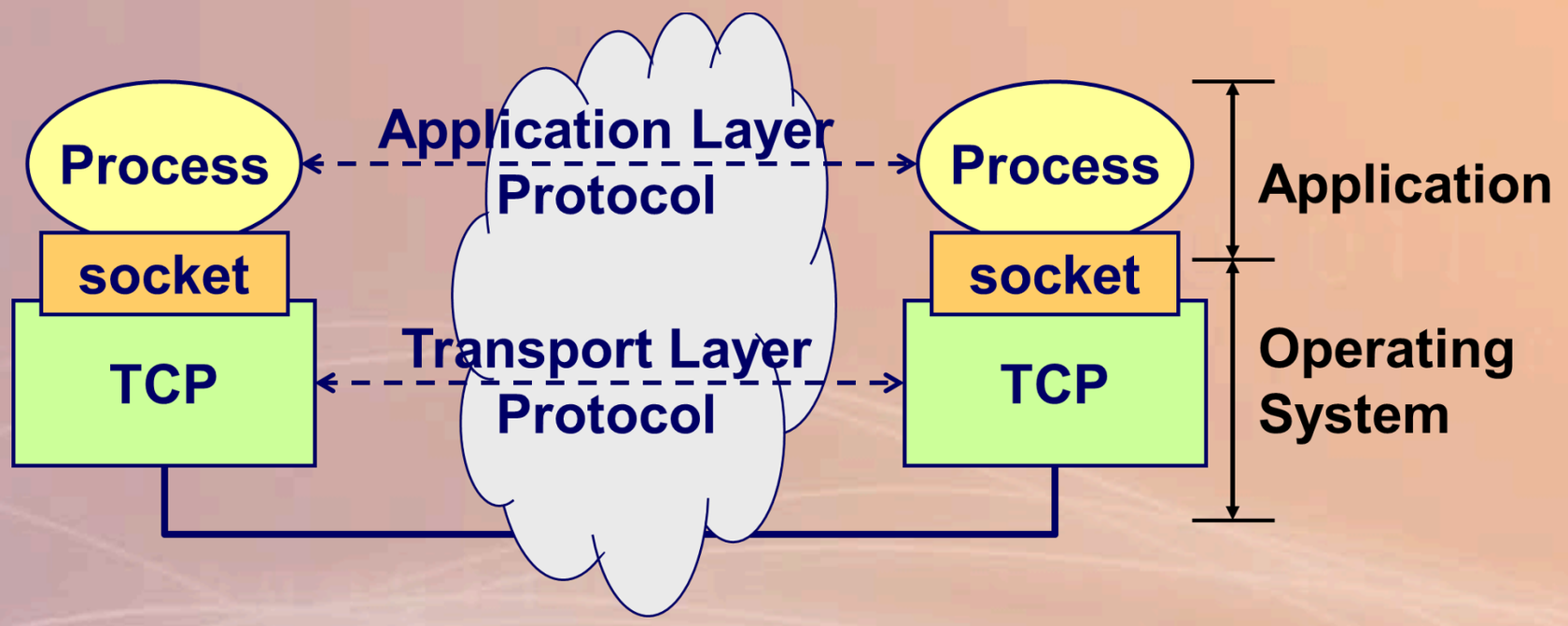
Messages and their processing define an application layer protocol.

What Defines an Application Protocol

- Message types
  - Syntax: How are messages formatted?
  - Semantics: What do messages mean?

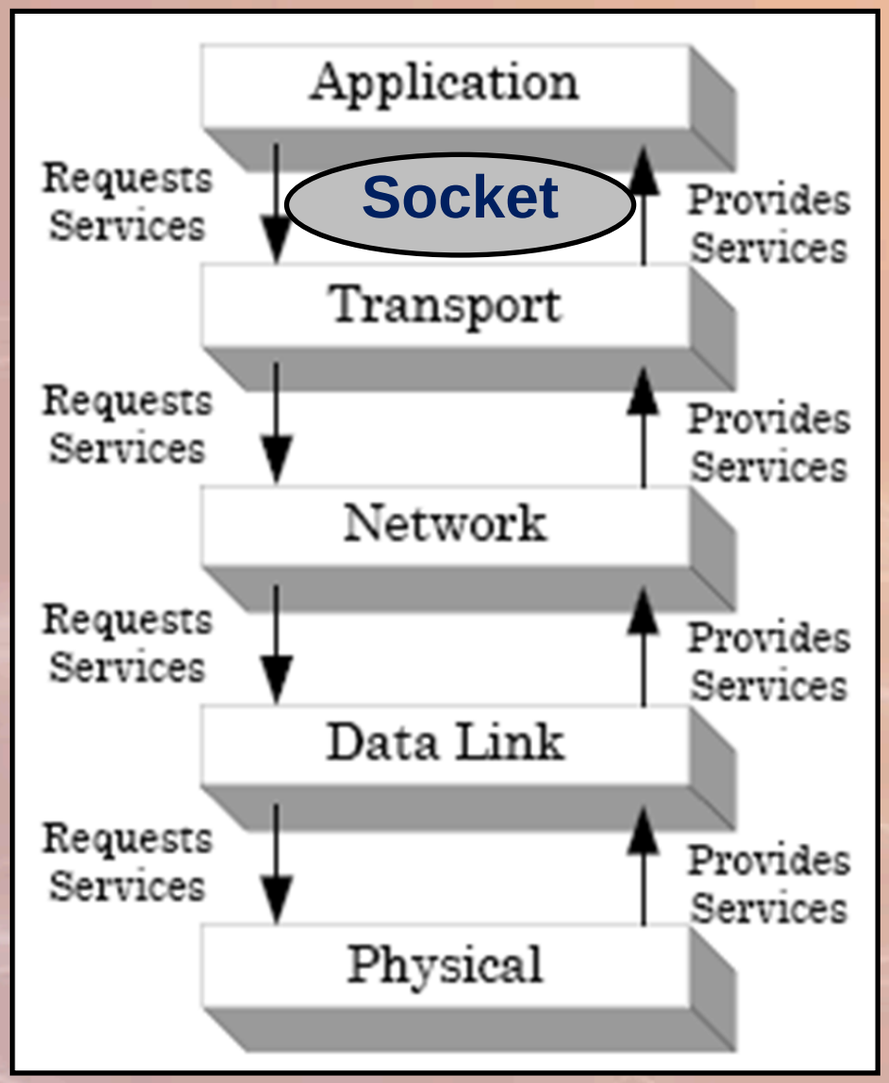
- Message processing
  - What actions are taken in response to messages?
  - What message sequences are valid?
  - What is the response in the event of errors?

Application Layers and Lower Processes



- Processes use an API to access operating system services for network communication
- Socket abstraction is widely used

Sockets as API



Sockets provide a software interface (API) between an application and a lower layer service provider.

File Descriptor

Remember our Unix file server example:

- Request: OPEN(/tmp/test.txt, “r”)
- Reply: OPEN(ok, 32) -- handle = 32

Unix file descriptor – represents a connection to a file through which you can access the file.

- fd = open(“hello.txt”,’r’)
- fd.read()

Sockets provide a way to speak to other programs (across the network) using standard Unix file descriptors.

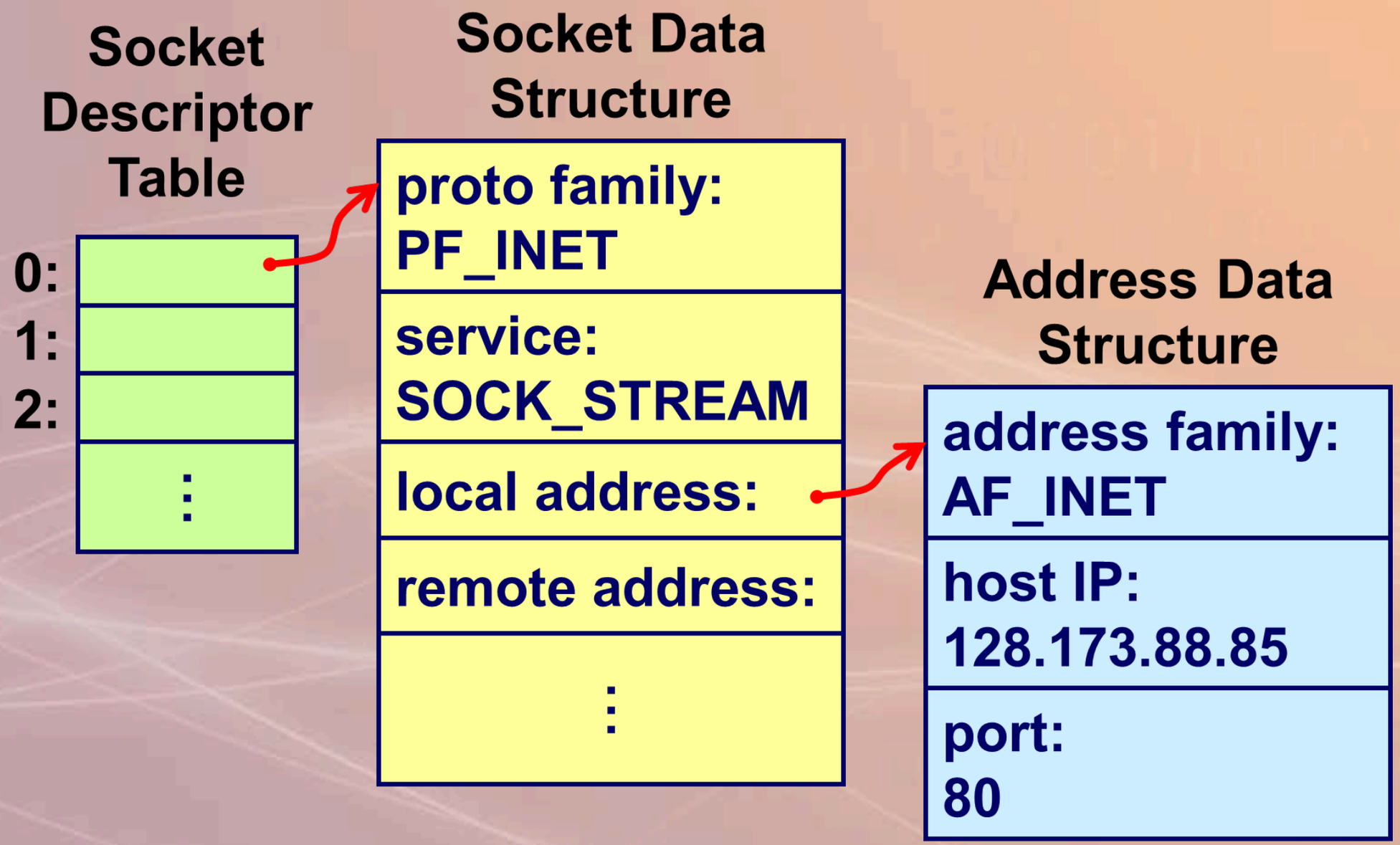
Sockets are file descriptors.

Socket Descriptors (1)

- Operating system maintains a set of socket descriptors for each process
  - Note that socket descriptors are shared by threads

- Three data structures
  - Socket descriptor table
  - Socket data structure
  - Address data structure

Socket Descriptors (2)



Python Socket Module

The Python *socket* module provides direct access to the standard Berkeley Unix socket interface

Communication through a socket descriptor is performed using specialized socket calls like **send()** and **recv()**.

```
import socket
  •
  •
sd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  •
sd.send(“Hello, World”)
  •
data = sd.recv(1024)
```

Server - Sockets

To create a server, you need to:

1. create a socket
2. bind the socket to an address and port
3. listen for incoming connections
4. wait for clients
5. accept a client
6. send and receive data

Client - Sockets

To create a client, you need to:

1. create a socket
2. connect to the server
3. send and receive data

Address and Protocol Families

```
sd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

First argument represents address and protocol families.

- AF_UNIX – Unix socket allows two processes running on the same machine to communicate with each other through the socket interface.
- AF_INET – A socket between two processes, typically running on different machines, using IP version 4 (IPv4) – 32-bit addresses
- AF_INET6 – Socket similar to IPv4 socket except that it uses IP version 6 (IPv6) – 128-bit addresses

Socket Types

```
sd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Second argument represents socket types.

- SOCK_STREAM
  - Reliable two-way connected communication streams.
  - Based on TCP
  - telnet uses stream sockets

- SOCK_DGRAM
  - Unreliable, connectionless communication streams.
  - Based on UDP
  - tftp uses datagram sockets

Socket Programming with TCP

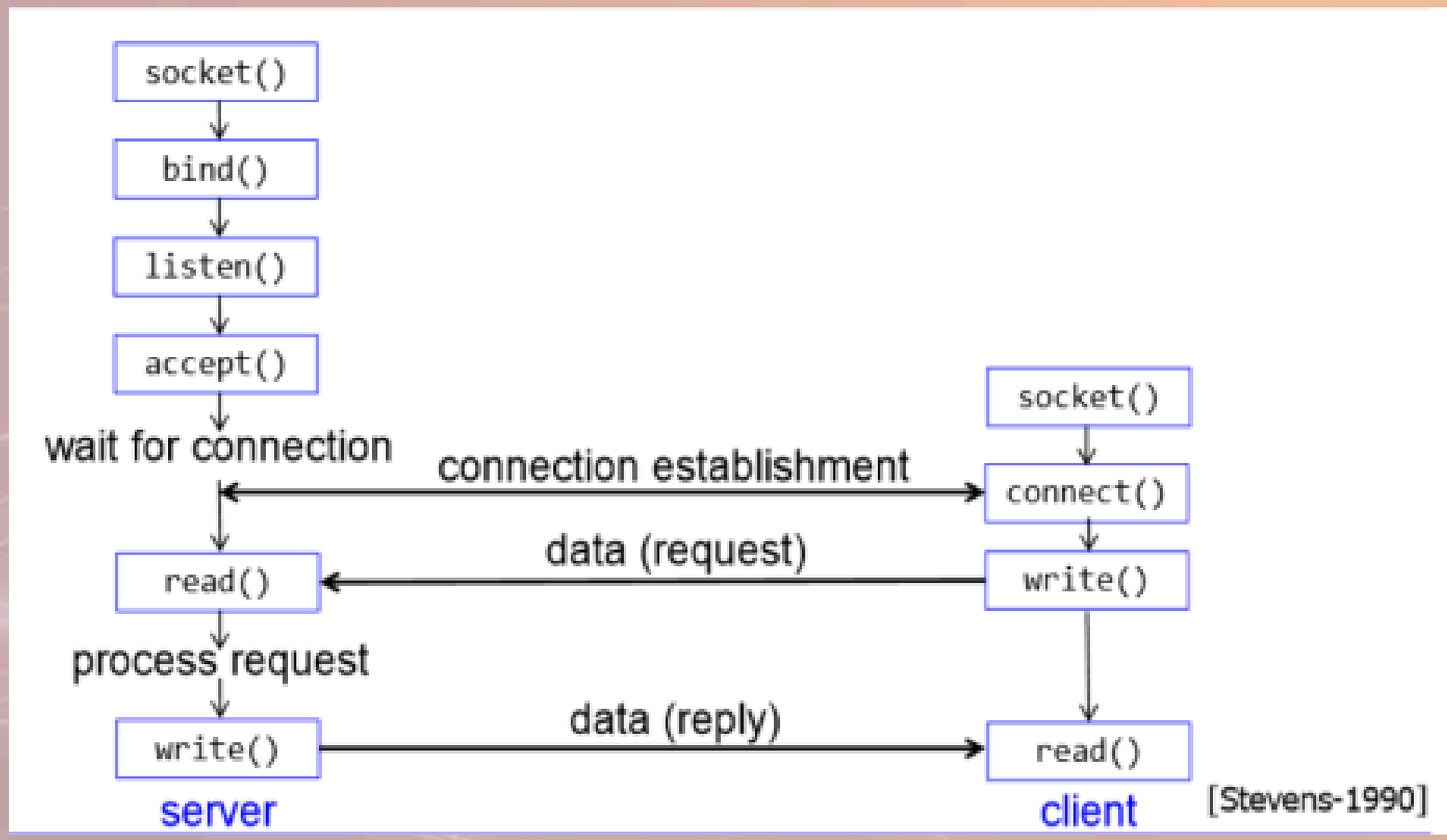
- Connect-oriented transport between client and server
  - Three-way handshake
  - Server must start first

- Connection reliable and ordered

- Client process
  - Creates socket of type StreamSocket

- Server process
  - Creates socket of type StreamSocket

Connection-Oriented Sockets



Socket Programming with UDP

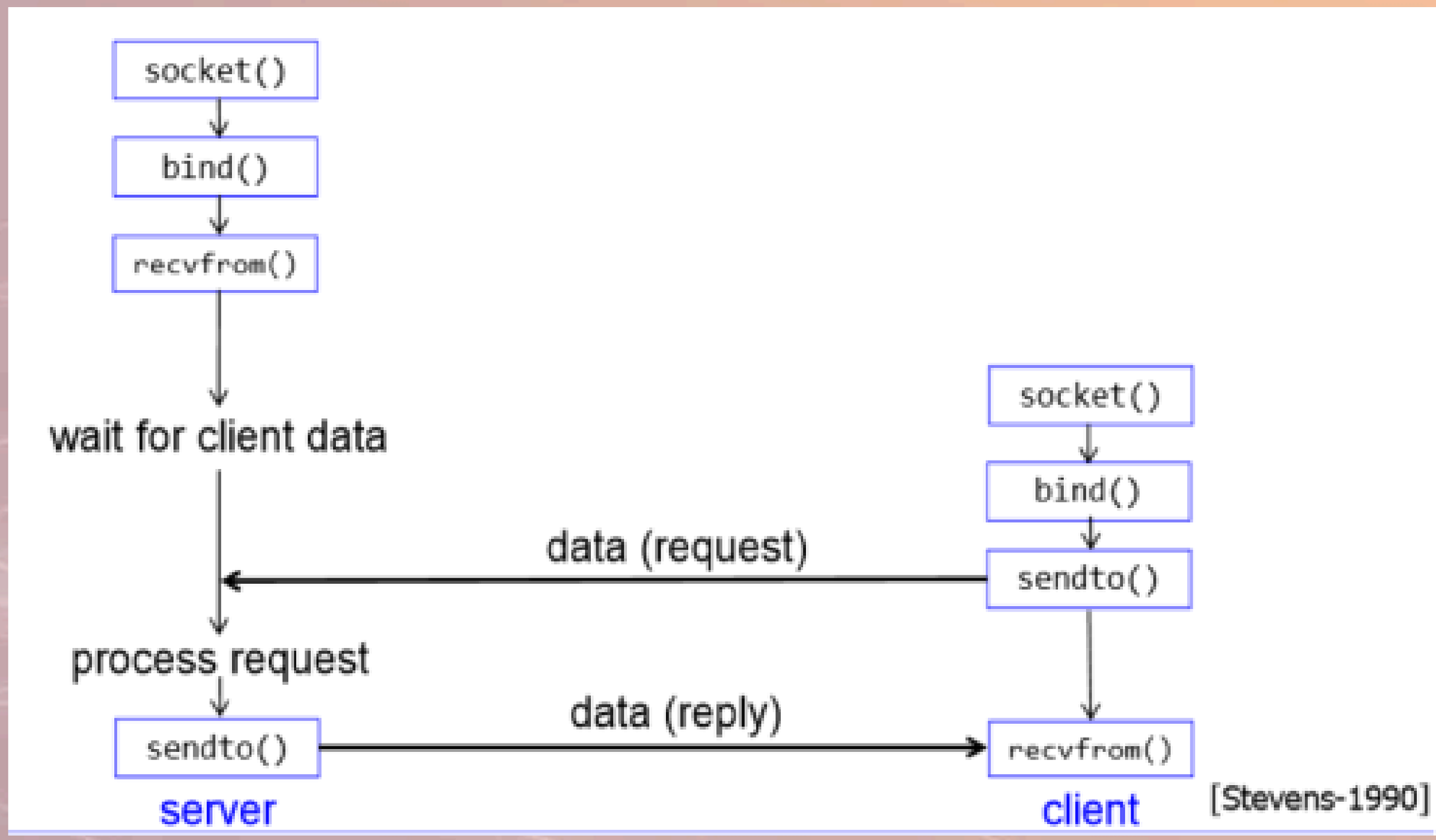
- Connectionless transport between client and server
  - No initial handshaking
  - Unlike TCP, client can be started first

- Client attaches destination address to each packet

- Client process
  - Creates clientSocket of type DatagramSocket

- Server process
  - Creates serverSocket of type DatagramSocket

Connectionless Sockets



Simple Echo Server

```
#!/usr/bin/env python

A simple echo server

import socket

host = ''
port = 50000
backlog = 5
size = 1024
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host,port))
s.listen(backlog)
while 1:
    client, address = s.accept()
    data = client.recv(size)
    if data:
        client.send(data)
    client.close()
```

Simple Echo Server

#!/usr/bin/env python

In a Unix-like operating system, the program loader takes the presence of "#!“ as an indication that the file is a script, and tries to execute that script using the interpreter specified by the rest of the first line in the file.

Simple Echo Server

```
import socket

host = ''
port = 50000
backlog = 5
size = 1024
```

- server imports the socket module
- sets the host name and the port number
- Initializes other variables

https://docs.python.org/2/library/socket.html

Simple Echo Server

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

This creates a socket object, stored in s, that will use IPv4 and TCP.

```
s.bind((host,port))
```

Binds the socket to the host name (the current host) and the port number specified above. This allows your application to accept incoming connections that designate your host address and the indicated port number as the server. Note that this call will fail if some other application is already using this port number on the same machine.

```
s.listen(backlog)
```

Listen for connections made to the socket. The backlog argument specifies the maximum number of queued connections and should be at least 0; the maximum value is system-dependent (usually 5). This means that you can have at most five clients waiting while the server is handling the current client.

Simple Echo Server

```
while 1:
    client, address = s.accept()
    data = client.recv(size)
    if data:
        client.send(data)
    client.close()
```

- Infinite loop accepts a client, reads a single message, and echoes that message back to client.
- accept() method accepts an incoming connection from a client.
  - Call returns a pair of arguments: *client, address*
  - *client* is a new socket object used to communicate with the client.
  - *address* is the address of the client.

Simple Echo Server

```
while 1:
    client, address = s.accept()
    data = client.recv(size)
    if data:
        client.send(data)
    client.close()
```

- The socket recv() method takes the maximum amount of data, in bytes, that you are willing to receive.
- The socket send() method sends data to the client.
  - Note that the return value of send() is the number of bytes actually sent.
- The socket close() method will close the socket.
  - All future operations on the socket will fail.

Simple Echo Client

```
#!/usr/bin/env python

"""
A simple echo client
"""

import socket

host = 'localhost'
port = 50000
size = 1024
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host,port))
s.send(b'Hello, world')
data = s.recv(size)
s.close()
print ('Received:', data)
```

Simple Echo Client

```
import socket

host = 'localhost'
port = 50000
```

- Import the *socket* module
- Set host name
  - Name of server providing service
- Set port number
  - Port server is listening on

Simple Echo Client

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

This creates a socket object, stored in s – same syntax as the server.

```
s.connect((HOST, PORT))
```

The client uses the connect() method to connect to the server..

```
s.send(b'Hello, world')
data = s.recv(size)
```

The *send()* method transmits data to the server. Returns how much data was actually sent.

Client uses *recv()* method to retrieve data from the server. Argument *size* (buffer) indicates the maximum amount of data it will handle at a time.

Simple Echo Client

```
s.close()
print ('Received:', data)
```

After data received, the socket is closed.

Received data is printed out.

Server continues to run.

Catching Exceptions on Server

Replace

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host,port))
s.listen(backlog)
while 1:
```

With

```
s = None
try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind((host,port))
    s.listen(backlog)
except socket.error, (value,message):
    if s:
        s.close()
    print "Could not open socket: " + message
    sys.exit(1)
while 1:
    …
```

Catching Exceptions on Client

```
#!/usr/bin/env python

# A simple echo client that handles some exceptions

import socket
import sys

host = 'localhost'
port = 50000
size = 1024
s = None
try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((host,port))
except socket.error, (value,message):
    if s:
        s.close()
    print "Could not open socket: " + message
    sys.exit(1)
s.send('Hello, world')
data = s.recv(size)
s.close()
print 'Received:', data
```

You should now know ...

- The role and functionality of a socket and a socket descriptor
- The basic operation of a socket-based client and server for a connection-oriented protocol
- Basic exception handling

Next Lecture

- Raspberry Pi GPIO
- Connectionless socket example
- More socket constructs
  - Select module
  - Threading module
- Assignment 1

Closing