

NETWORK PROGRAMMING USING PYTHON

CUAUHTEMOC CARBAJAL
ITESM CEM
APRIL 06, 2013

NETWORK OVERVIEW

- Network Overview
 - Python provides a wide assortment of network support
 - Low-level programming with sockets (if you want to create a protocol).
 - Support for existing network protocols (HTTP, FTP, SMTP, etc...)
 - Web programming (CGI scripting and HTTP servers)
 - Data encoding
- I can only cover some of this
 - Programming with sockets (this lecture)
 - HTTP and Web related modules (next lecture: Internet Client Programming using Python)
 - A few data encoding modules (next lecture)
- Recommended Reference
 - Unix Network Programming by W. Richard Stevens.

THE INTERNET PROTOCOL

- Both networking,
 - which occurs when you connect several computers together so that they can communicate,
- and internetworking,
 - which links adjacent networks together to form a much larger system like the Internet,
- are essentially just elaborate schemes to allow resource sharing.
 - All sorts of things in a computer need to be shared: disk drives, memory, the CPU, and of course the network.
 - The physical networking devices that your computer uses to communicate are themselves each designed with an elaborate ability to share a single physical medium among many different devices that want to communicate.
- The fundamental unit of sharing among network devices is the “packet.”
 - binary string whose length might range from a few bytes to a few thousand bytes, which is transmitted as a single unit between network devices.
 - has only two properties at the physical level:
 - the binary string that is the data it carries, and an address to which it is to be delivered.

WHAT, THEN, IS THE INTERNET PROTOCOL?

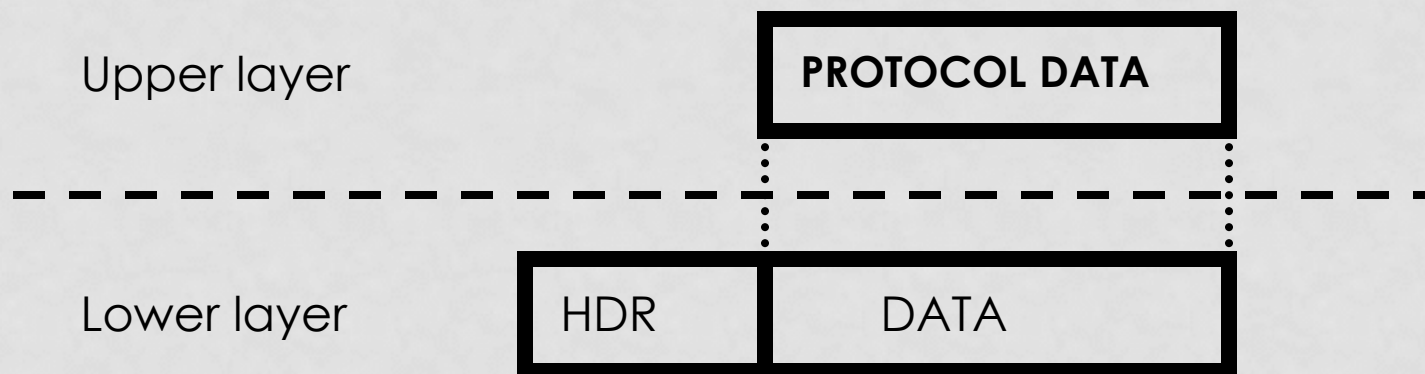
- The Internet Protocol is:
 - a scheme for imposing a uniform system of addresses on all of the Internet-connected computers in the entire world, and
 - to make it possible for packets to travel from one end of the Internet to the other.
- Ideally, an application like your web browser should be able to connect a host anywhere without ever knowing which maze of network devices each packet is traversing on its journey.
- It is very rare for a Python program to operate at such a low level that it sees the Internet Protocol itself in action, but in many situations, it is helpful to at least know how it works.

NETWORK LAYERING

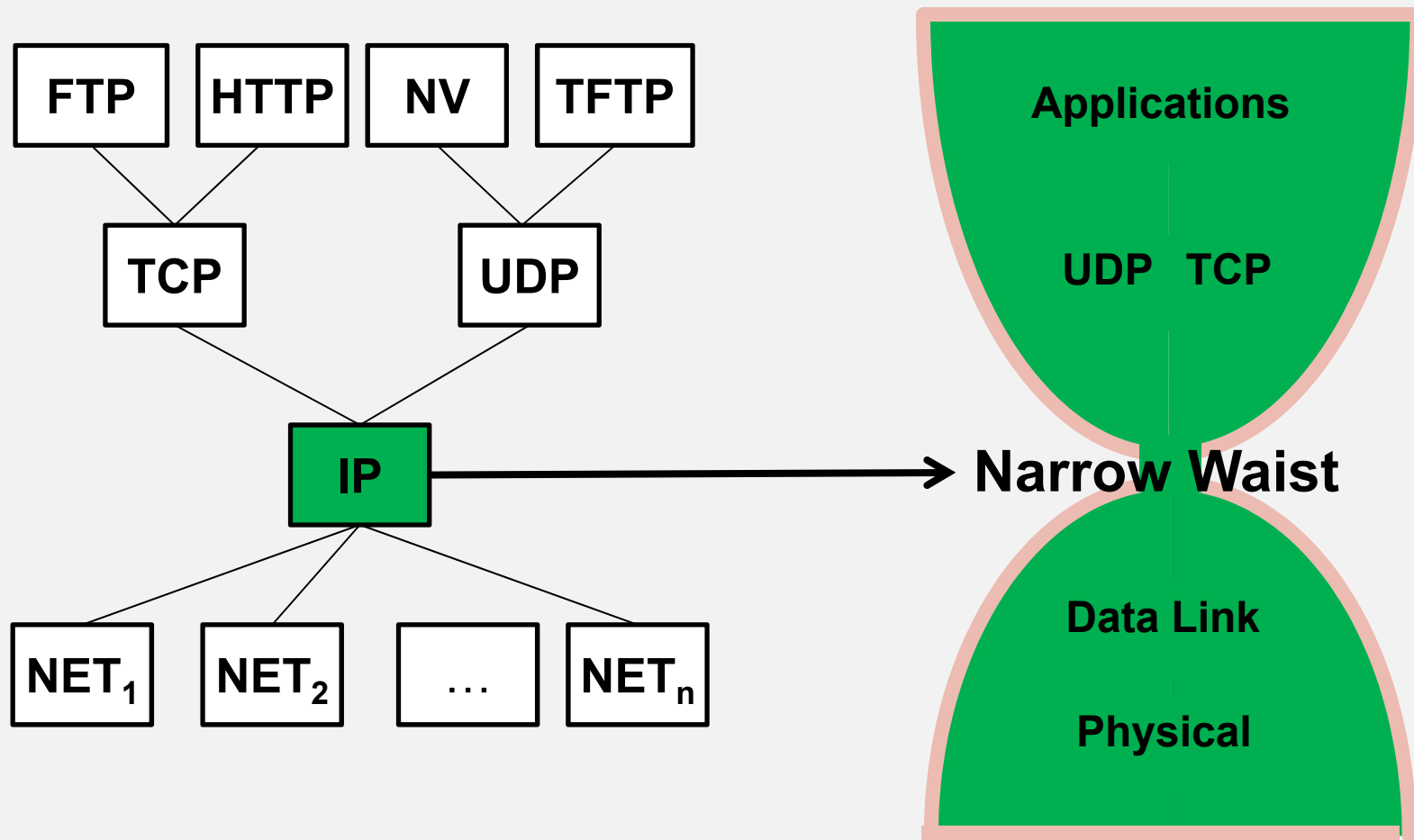
- Applications talk to each other
 - Call transport layer functions
- Transport layer has to ship packets
 - Calls network layer
- Network layer talks to next system
 - Calls subnetwork layer
- Subnetwork layer frames data for transmission
 - Using appropriate physical standards
 - Network layer datagrams "hop" from source to destination through a sequence of routers

INTER-LAYER RELATIONSHIPS

- Each layer uses the layer below
 - The lower layer adds headers to the data from the upper layer
 - The data from the upper layer can also be a header on data from the layer above ...



THE TCP/IP LAYERING MODEL



The Hourglass Model ⁷

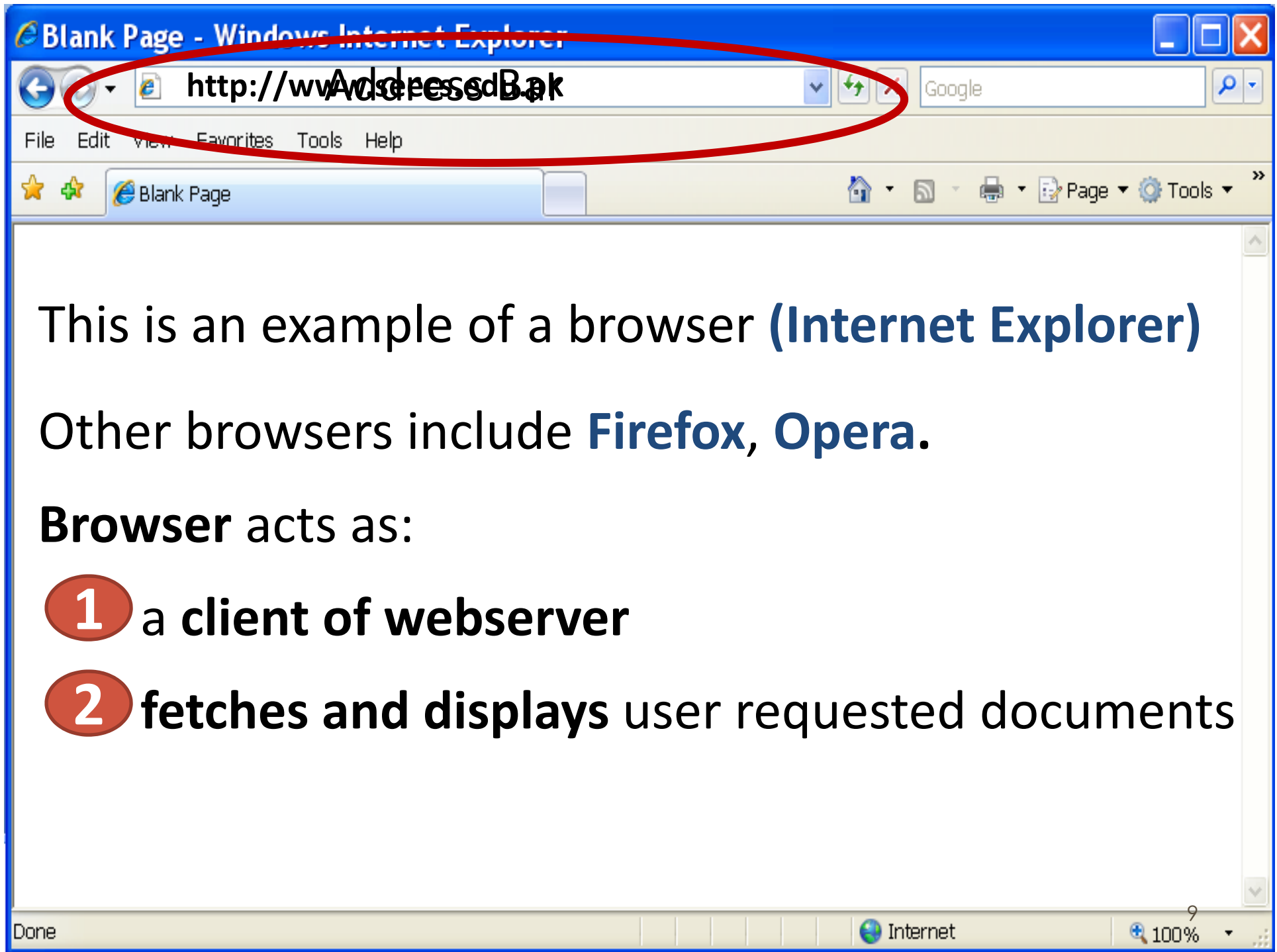
Example application

To clarify concepts, let us assume
a greatly simplified model of the
LAN of NUST-SEECs

Network's domain name:
seecs.edu.pk

Let's assume a

seecs.edu.pk LAN) will access NUST-
SEECs website hosted at **www.seecs.edu.pk**



This is an example of a browser (**Internet Explorer**)

Other browsers include **Firefox**, **Opera**.

Browser acts as:

- 1** a client of webserver
- 2** fetches and displays user requested documents

Example application (contd.)

The **HTTP** request sent by the student PC (the machine pc.seecs.edu.pk) to the webserver (the machine www.seecs.edu.pk) would be something like “**GET / HTML/1.1**”

Packet so far: GET / HTML/1.1

Outstanding issues:

- 1 How to send this request to Webserver?
- 2 Which application at webserver must process this packet?

Example application (contd.)

1 But how to send this request to Webserver?

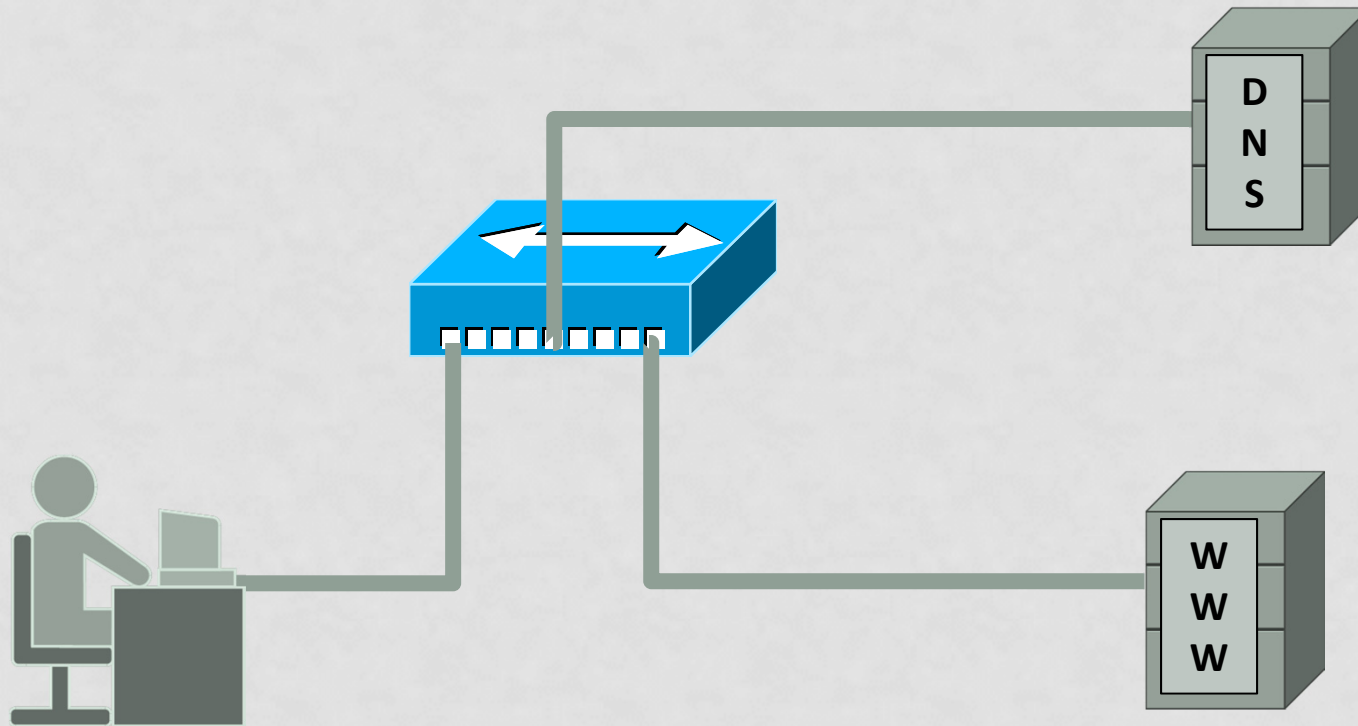
To communicate with www.seecs.edu.pk
(**hostname**), its IP **address** must be known

How to resolve **hostnames** to IP **addresses**

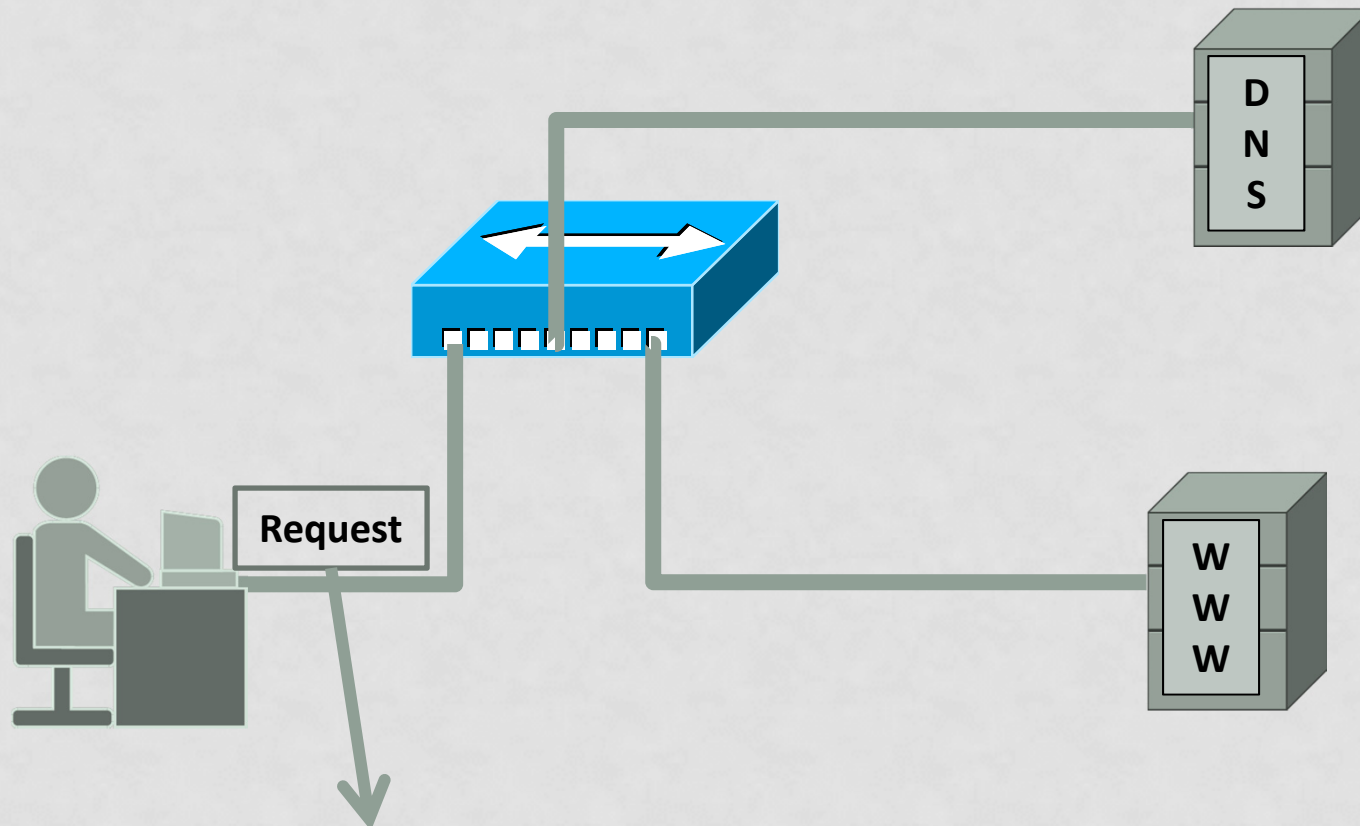


Domain Name Service (DNS)

Example application (contd.)

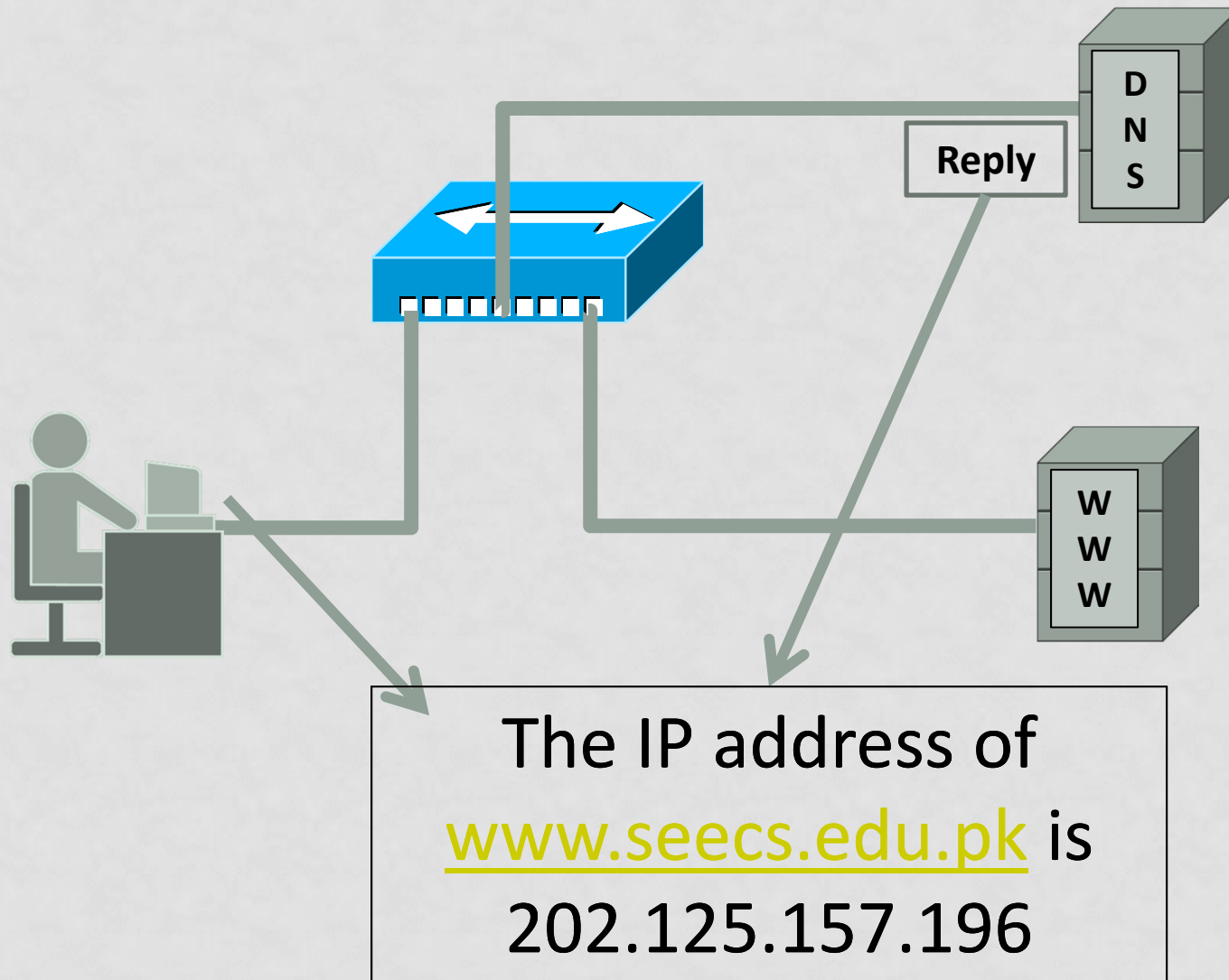


DNS Client/Server Exchange



Tell me the IP address
of www.seecs.edu.pk?

DNS Client/Server Exchange



Example application (contd.)

2 Which application at webserver must process this packet?

In TCP/IP, each well-known application is identified using **ports**.

The port of DNS is **53**; HTTP is **80**; SMTP is **25**.

In our considered example, **HTTP server application (port 80)** would process the packet.

Packet so far:

Source Port	Destination Port	GET / HTML/1.1
> 1024	80	

Example application (contd.)

The destination IP address (found through DNS) is **202.125.157.196**.

Let's assume the source IP address is

202.125.157.150

(network must be same; to be explained later)

Packet so far:

Source IP	Destination IP	Source Port	Destination Port	GET / HTML/1.1
202.125.157.150	202.125.157.196	> 1024	80	

Logical addressing: **network** and **host** parts

*Assuming **/24** subnet mask (to be explained later)

Example application (contd.)

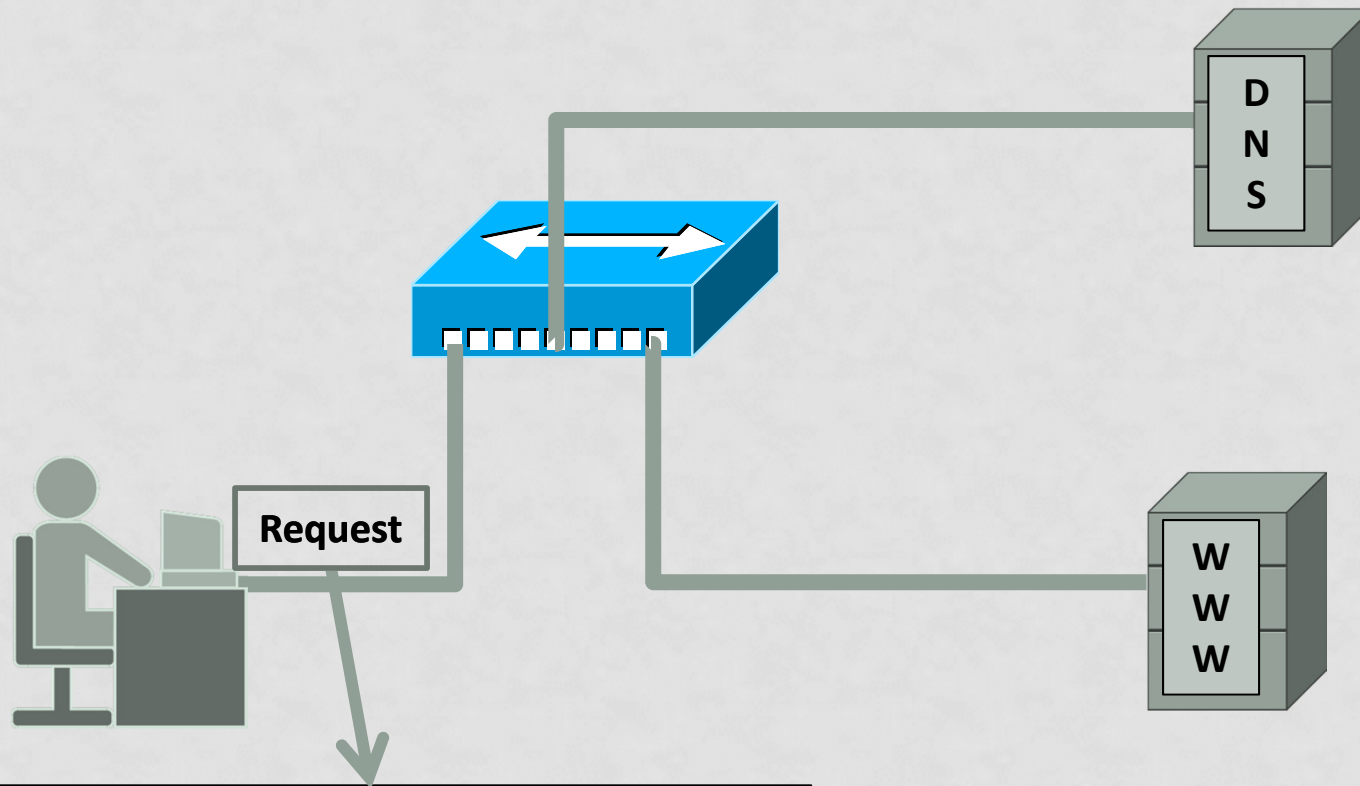
3 How to send the created **packet** to Webserver?

To communicate with any host, its physical address (called **MAC address**) must be known.

How to resolve **IP addresses** to **MAC addresses** 

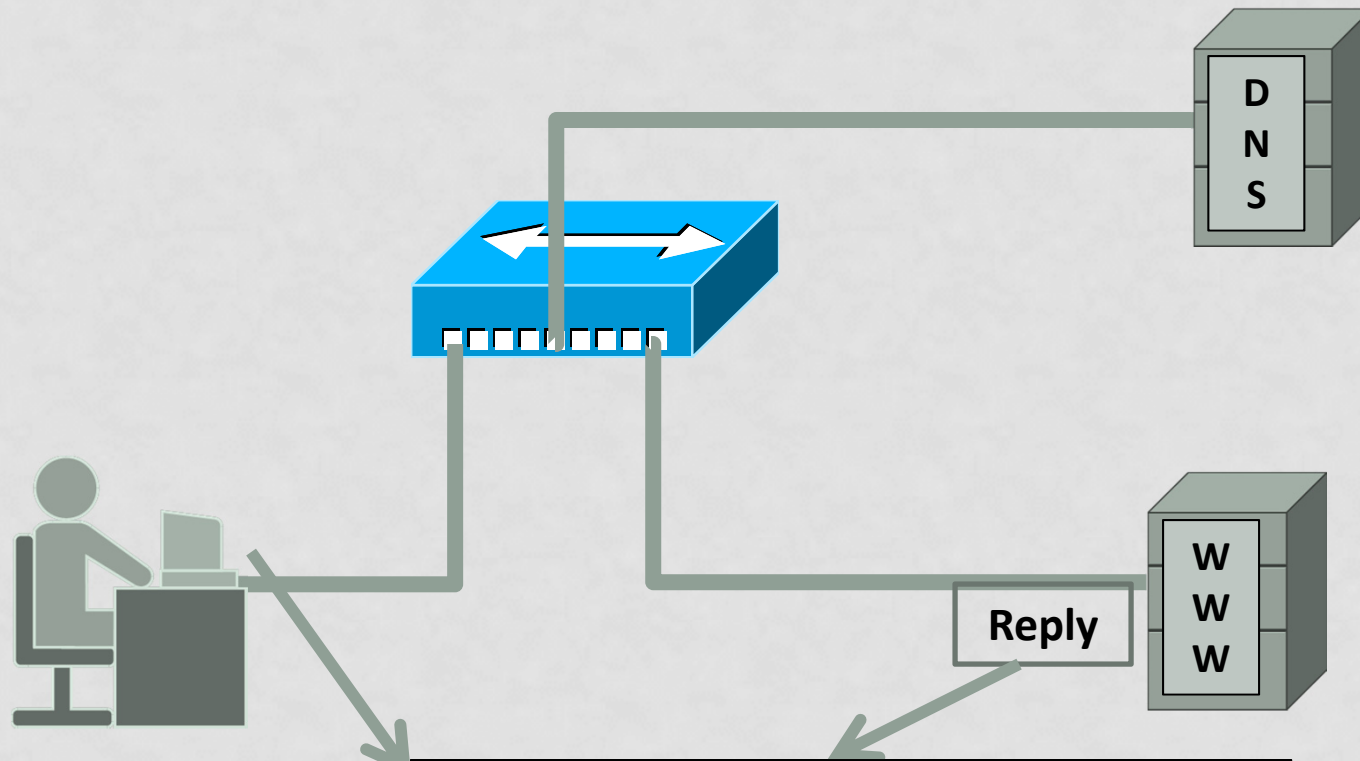
Address Resolution Protocol (ARP)

ARP Client/Server Exchange



Any one knows the
MAC (physical) address
of **202.125.157.196** ?

ARP Client/Server Exchange



The MAC address of
202.125.157.196 is
12:34:aa:bb:cc:dd

Example application (contd.)

Now that the physical (MAC) addresses are known, communication can take place

The destination MAC address is **12:34:aa:bb:cc:dd**

The source MAC address (let's assume) is **23:34:aa:bb:cc:dd**

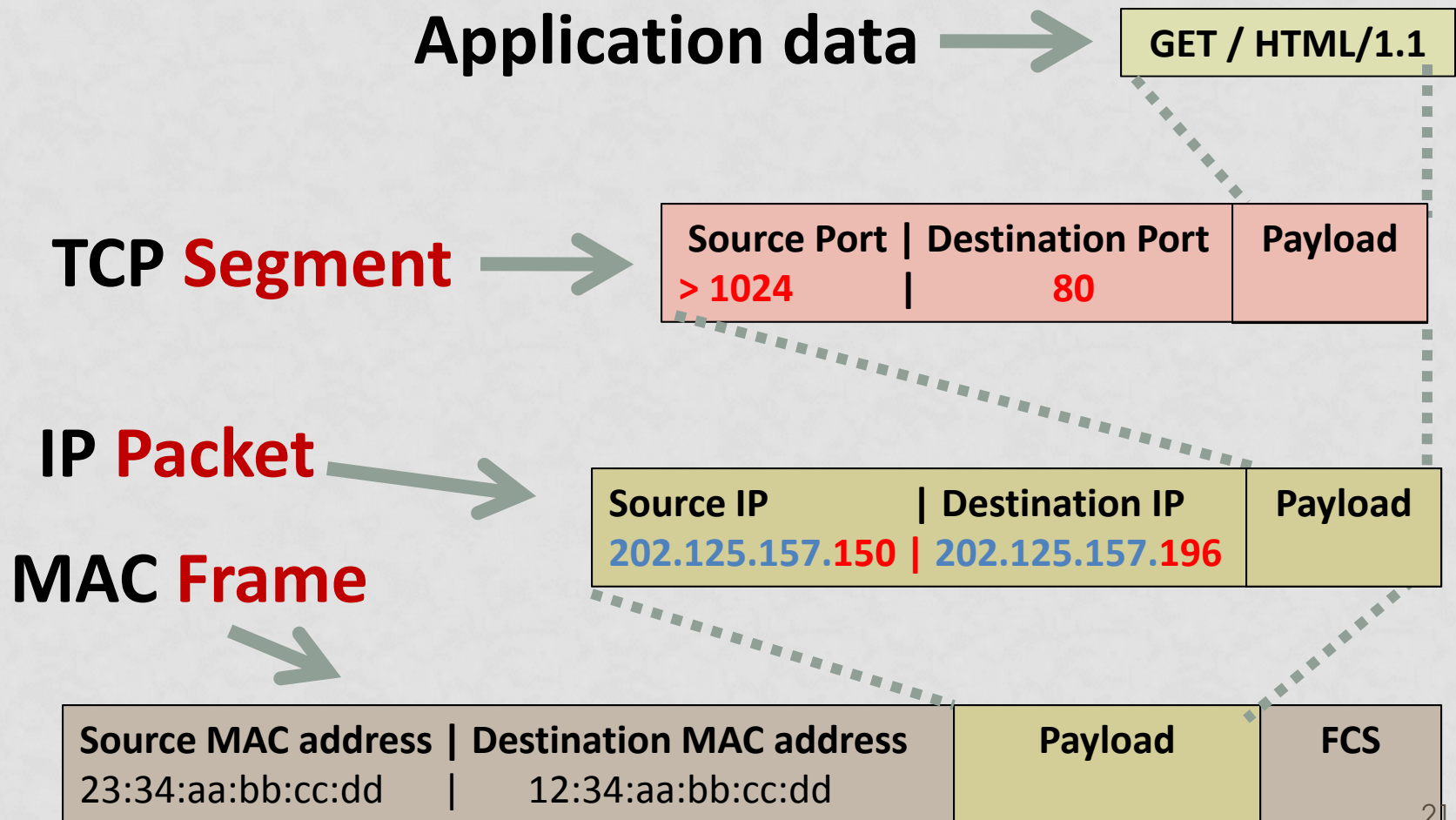
IP packet containing the data

Source IP	Destination IP	Source Port	Destination Port	GET / HTML/1.1
202.125.157.150	202.125.157.196	> 1024	80	

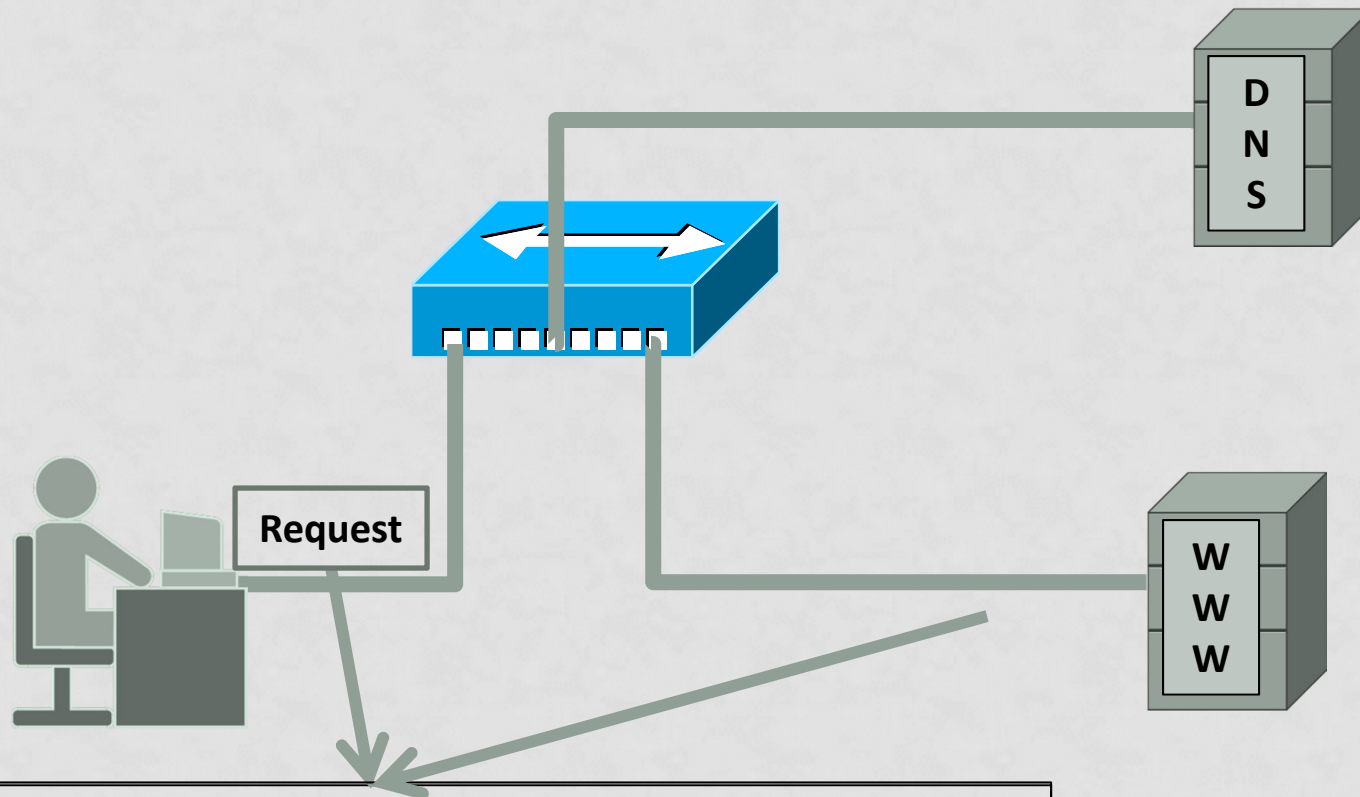
MAC frame

Source MAC address	Destination MAC address	Payload	FCS
--------------------	-------------------------	---------	-----

Encapsulation



HTTP Client/Server Exchange



Send me the index.html page
for the host www.seecs.edu.pk
using HTTP version 1.1

HTTP Client/Server Exchange

NUST School of Electrical Engineering and Computer Science (NUST-SEECs) - Service Pack 3 Internet Explorer

File Edit View Favorites Tools Help

Back Forward Stop Reload Home Search Favorites RSS Print Mail News Groups

Address <http://www.seecs.edu.pk/> Go Links



National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

A Center of Excellence for Quality Education and Research

NUST
Defining Futures

[LMS](#) [Home](#) [Contact Us](#)

Quick Links

[Library](#) | [Career](#) | [Directory](#) | [Site Map](#) | [E-Mail](#)

- Home
- About Us
- Academics
- Admissions
- Research Groups
- Campus Life
- Student Resources
- Exam Branch
- Library
- Related Websites
- Research Websites
- Sub-Websites
- Downloads
- Alumni

News and Events

Admission for Undergraduate Programmes

Applications are invited for admission to undergraduate programmes of NUST for Session 2009.
[For detail click here](#)

NUST SEECs got another feather in its cap on the 62nd independence anniversary of Pakistan, with the conferment of the prestigious award of 'Pride of Performance' on Professor Dr Arshad Ali and Dr Mohammad Hassan Zaidi, DG and Dean SEECs respectively. This gives a new fillip to the pride and inspiration of everyone associated with SEECs.

SEECs has secured the honor of becoming first HL7 Organization Member from Pakistan. Board of Directors of Health Level Seven awarded the SEECs the membership.

Highlights



Event Calendar

« April 2009 »

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

Notice Board

In order to provide better transport facilities to NUST Student/Staff, NUST has decided to start shuttle service for those who can not avail regular NUST transport.
[Click Here for Detail](#)

Class Schedule for UG and PG Classes
[Click Here for Detail](#)



Night view of NUST SEECs Building

23

NUST School of Electrical Engineering and Computer Science (NUST-SEECs) Internet

NETWORK BASICS: TCP/IP

- When a network application is built on top of IP, its designers face a fundamental question:
 - Will the network conversations in which the application will engage best be constructed from individual, unordered, and unreliable network packages?
 - Or will their application be simpler and easier to write if the network instead appears to offer an ordered and reliable stream of bytes, so that their clients and servers can converse as though talking to a local pipe?
- There are two basic possible approaches to building atop IP.
 - The vast majority of applications today are built atop TCP, the **Transmission Control Protocol**, which offers ordered and reliable data streams between IP applications.
 - A few protocols, usually with short, self-contained requests and responses, and simple clients that will not be annoyed if a request gets lost and they have to repeat it, choose UDP, the **User Datagram Protocol**.

IP CHARACTERISTICS

- Datagram-based
 - Connectionless
- Unreliable
 - Best efforts delivery
 - No delivery guarantees
- Logical (32-bit) addresses
 - Unrelated to physical addressing
 - Leading bits determine network membership

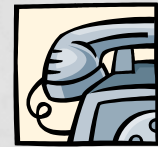
UDP CHARACTERISTICS

- Also datagram-based
 - Connectionless, unreliable, can broadcast
- Applications usually message-based
 - No transport-layer retries
 - Applications handle (or ignore) errors
- Processes identified by port number
- Services live at specific ports
 - Usually below 1024, requiring privilege



TCP CHARACTERISTICS

- Connection-oriented
 - Two endpoints of a virtual circuit
- Reliable
 - Application needs no error checking
- Stream-based
 - No predefined blocksize
- Processes identified by port numbers
- Services live at specific ports



UDP VERSUS TCP

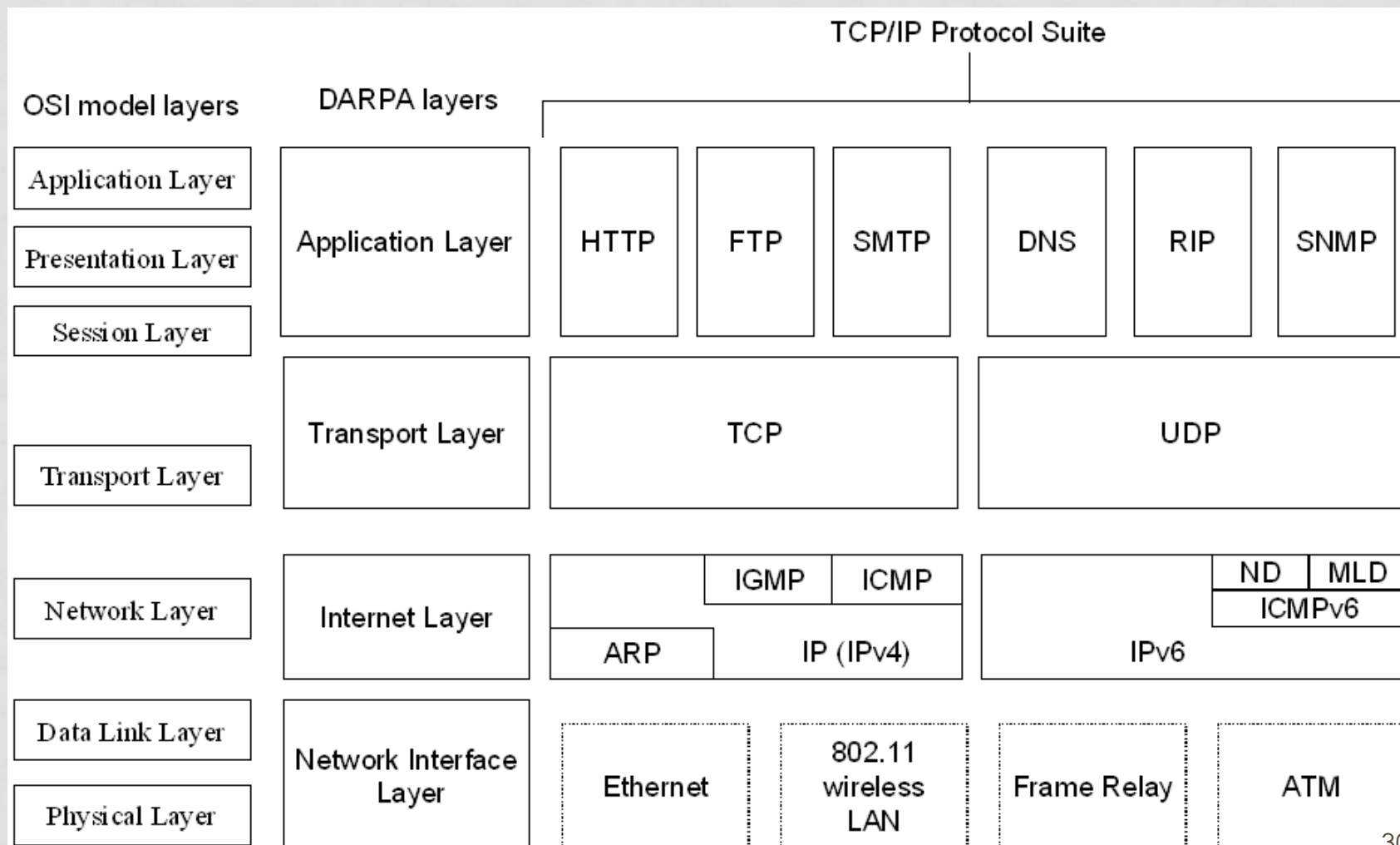
UDP v/s TCP		
Characteristics/ Description	UDP	TCP
General Description	Simple High speed low functionality "wrapper" that interface applications to the network layer and does little else	Full-featured protocol that allows applications to send data reliably without worrying about network layer issues.
Protocol connection Setup	Connection less data is sent without setup	Connection-oriented; Connection must be Established prior to transmission.
Data interface to application	Message base-based is sent in discrete packages by the application.	Stream-based; data is sent by the application with no particular structure
Reliability and Acknowledgements	Unreliable best-effort delivery without acknowledgements	Reliable delivery of message all data is acknowledged.
Retransmissions	Not performed. Application must detect lost data and retransmit if needed.	Delivery of all data is managed, and lost data is retransmitted automatically.
Features Provided to Manage flow of Data	None	Flow control using sliding windows; window size adjustment heuristics; congestion avoidance algorithms
Overhead	Very Low	Low, but higher than UDP
Transmission speed	Very High	High but not as high as UDP
Data Quantity Suitability	Small to moderate amounts of data.	Small to very large amounts of data.

TCP/IP COMPONENTS

- Just some of the protocols we expect to be available in a “TCP/IP” environment

Telnet	SSH	SMTP	FTP	NFS	DNS	SNMP	Application
TCP				UDP			Host-to-host
IP							Internetwork
Ethernet, Token Ring, RS232, IEEE 802.3, HDLC, Frame Relay, Satellite, Wireless Links, Wet String							Subnetwork

TCP/IP



COMPARISON BETWEEN STREAMING AND DOWNLOADING

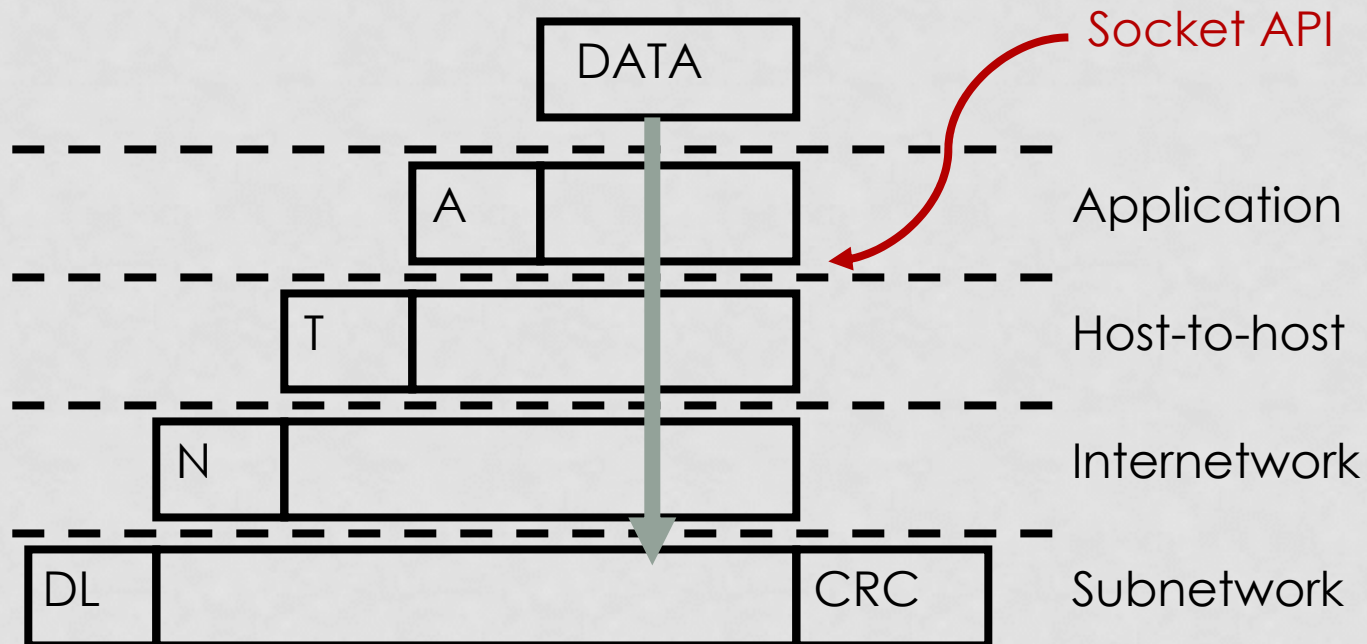
	Streaming	(Progressive) Download
Server	Streaming server is required	Standard web server is sufficient
Network layer protocol used	UDP/IP	TCP/IP
Application layer protocol used	RTP/RTSP	HTTP
Packet loss	Packet loss acceptable	No packet loss
Time performance	Real time. The delivered media duration is the same as original	Packets may be retransmitted, leading to slower delivery times
Delivery quality	Some packets may be discarded, to meet time and/or bandwidth constraints	High-quality delivery guaranteed, no data is lost or discarded
User connection	Can match the user's bandwidth	File is delivered without regard to the user's bandwidth
Playback	File starts playing immediately	Playback begins when all of (in progressive: enough of) the file has been downloaded
Effort	More burden on service provider (requires server, multiple bit-rate versions and formats)	More burden on the end user (hard drive space, connection speed)
Firewalls	May not play behind some firewalls	Bypasses most firewalls
Storage	No files are downloaded to the user's PC	Files are downloaded to the user's PC
VCR functionalities	Yes (for streaming of pre-recorded material)	No
Zapping of internet radio channels	Smooth	Not possible

SOCKETS

- Both protocols (UDP, TCP) are supported using "sockets"
 - A socket is a file-like object.
 - Allows data to be sent and received across the network like a file.
 - But it also includes functions to accept and establish connections.
 - Before two machines can establish a connection, both must create a socket object.

THE TCP/IP LAYERING MODEL

- Simpler than OSI model, with four layers



NETWORK BASICS: PORTS

- Ports

- In order to receive a connection, a socket must be bound to a port (by the server).
- A port is a number in the range 0-65535 that's managed by the OS.
- Used to identify a particular network service (or listener).
- Ports 0-1023 are reserved by the system and used for common protocols

FTP	Port 20
Telnet	Port 23
SMTP (Mail)	Port 25
HTTP (WWW)	Port 80

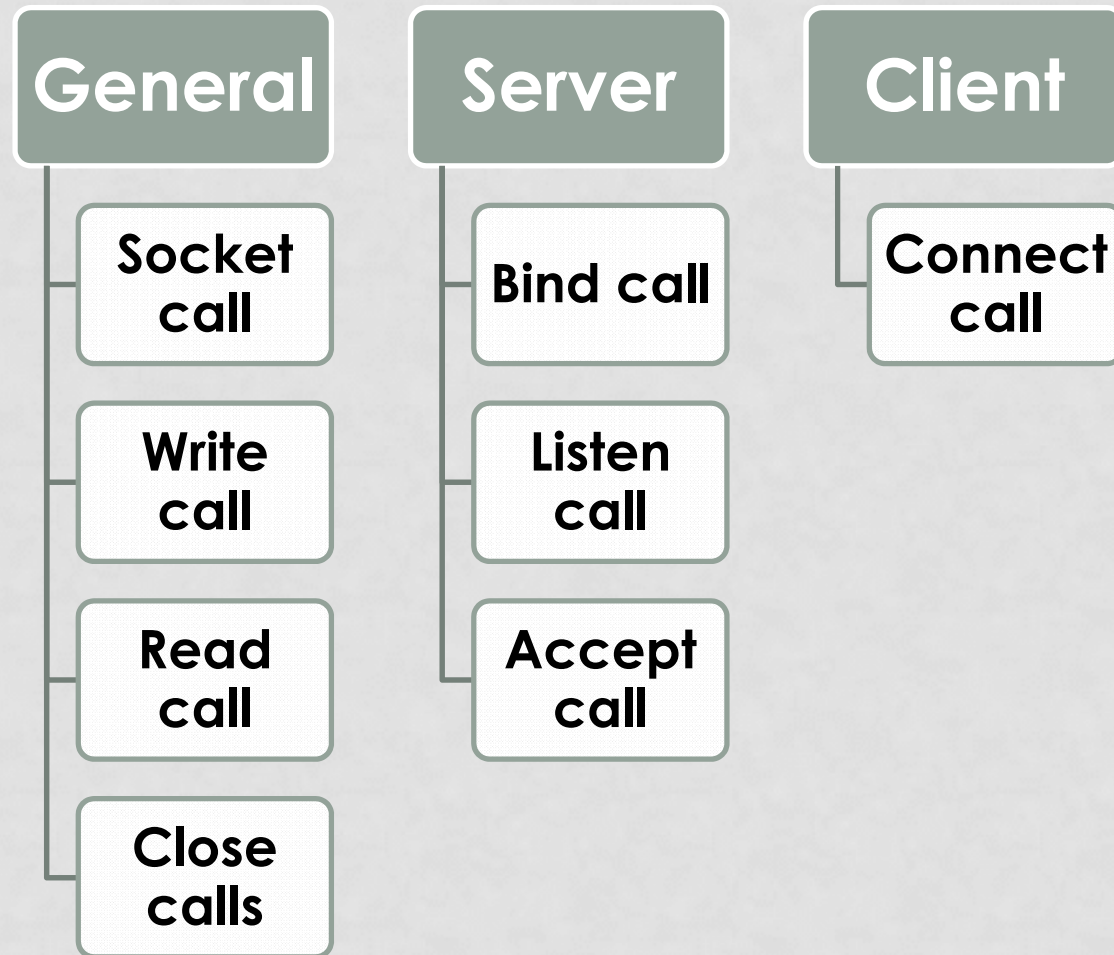
SSH	Port 22
DNS	Port 53

- Ports above 1024 are reserved for user processes.

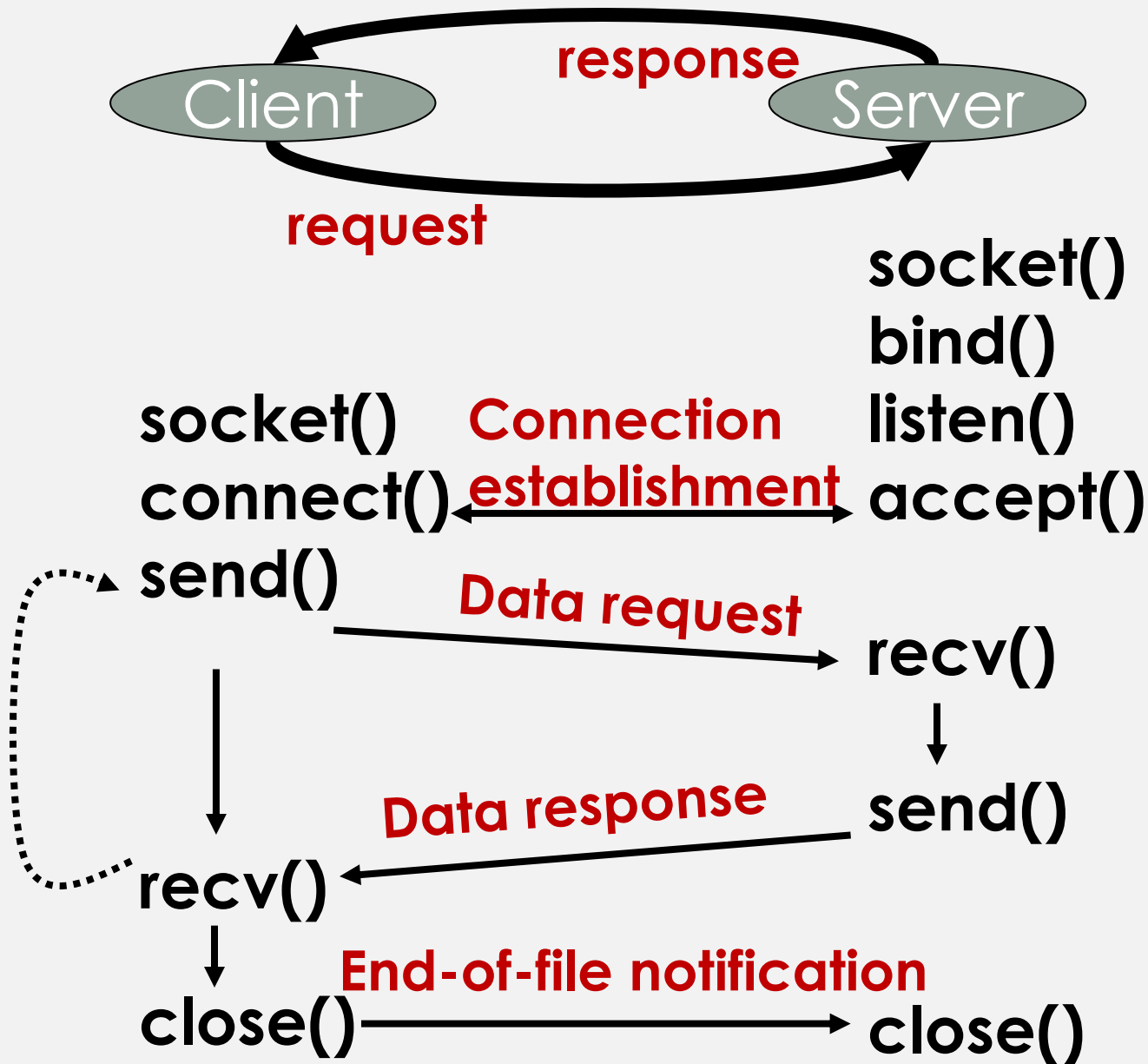
SOCKET PROGRAMMING IN A NUTSHELL

- Server creates a socket, binds it to some well-known port number, and starts listening.
- Client creates a socket and tries to connect it to the server (through the above port).
- Server-client exchange some data.
- Close the connection (of course the server continues to listen for more clients).

MAJOR SYSTEM CALLS



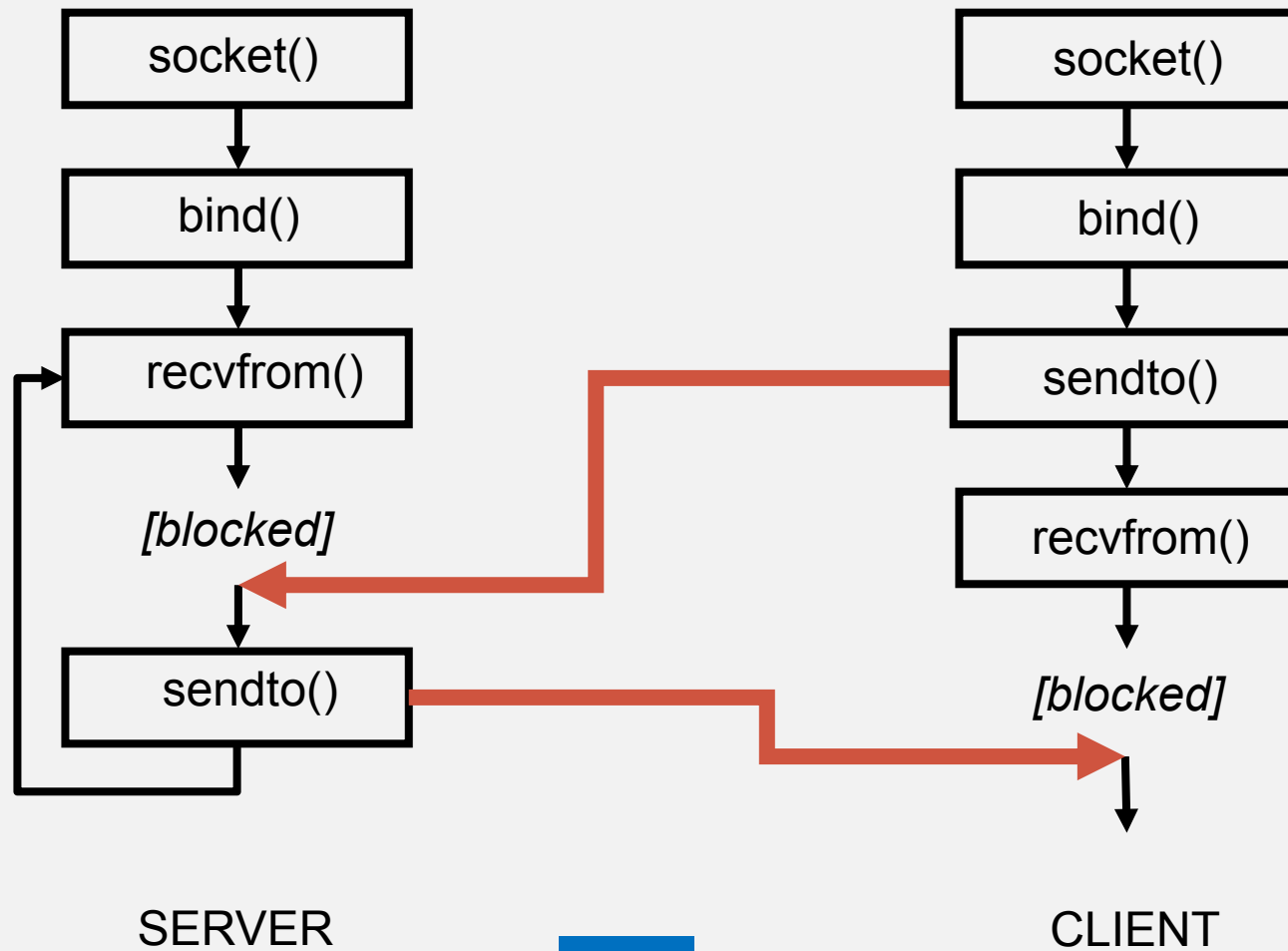
MAJOR SYSTEM CALLS



MAJOR SYSTEM CALLS (SUMMARY)

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Block the caller until a connection attempt arrives
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

CONNECTIONLESS SERVICES



SIMPLE CONNECTIONLESS SERVER

```
from socket import socket, AF_INET, SOCK_DGRAM
s = socket(AF_INET, SOCK_DGRAM)
s.bind(('127.0.0.1', 11111))
while 1:
    data, addr = s.recvfrom(1024)
    print "Connection from", addr
    s.sendto(data.upper(), addr)
```



- How much easier does it need to be?

Note that the *bind()* argument is a two-element tuple of address and port number

SIMPLE CONNECTIONLESS CLIENT

```
from socket import socket, AF_INET, SOCK_DGRAM
s = socket(AF_INET, SOCK_DGRAM)
s.bind(('127.0.0.1', 0))      # OS chooses port
print "using", s.getsockname()
server = ('127.0.0.1', 11111)
s.sendto("MixedCaseString", server)
data, addr = s.recvfrom(1024)
print "received", data, "from", addr
s.close()
```



- Relatively easy to understand?

EXERCISE 1: UDP CLIENT/SERVER

- Run the sample UDP client and server I have provided
 - `udpserv1.py`
 - `udpcli1.py`
- Additional questions:
 - How easy is it to change the port number and address used by the service?
 - What happens if you run the client when the server isn't listening?



UDP APPLICATION

- Problem: remote debugging
 - Need to report errors, print values, etc.
 - Log files not always desirable
 - Permissions issues
 - Remote viewing often difficult
 - Maintenance (rotation, etc.) issues
- Solution: messages as UDP datagrams
 - e.g. "Mr. Creosote" remote debugger
 - <http://starship.python.net/crew/jbauer/creosote/> (broken)
 - <http://hex-dump.googlecode.com/svn/trunk/tools/traceutil.py>
 - <https://gist.github.com/pklaus/974838>

<http://www.youtube.com/watch?v=aczPDGC3f8U>

CREOSOTE OUTPUT

```
def spew(msg, host='localhost', port=PORT):  
    s =  
    socket.socket((socket.AF_INET, socket.SOCK_DGRAM))  
    s.bind(('', 0))  
    while msg:  
        s.sendto(msg[:BUFSIZE], (host, port))  
        msg = msg[BUFSIZE:]
```

- Creates a datagram (UDP) socket
- Sends the message
 - In chunks if necessary

CREOSOTE INPUT

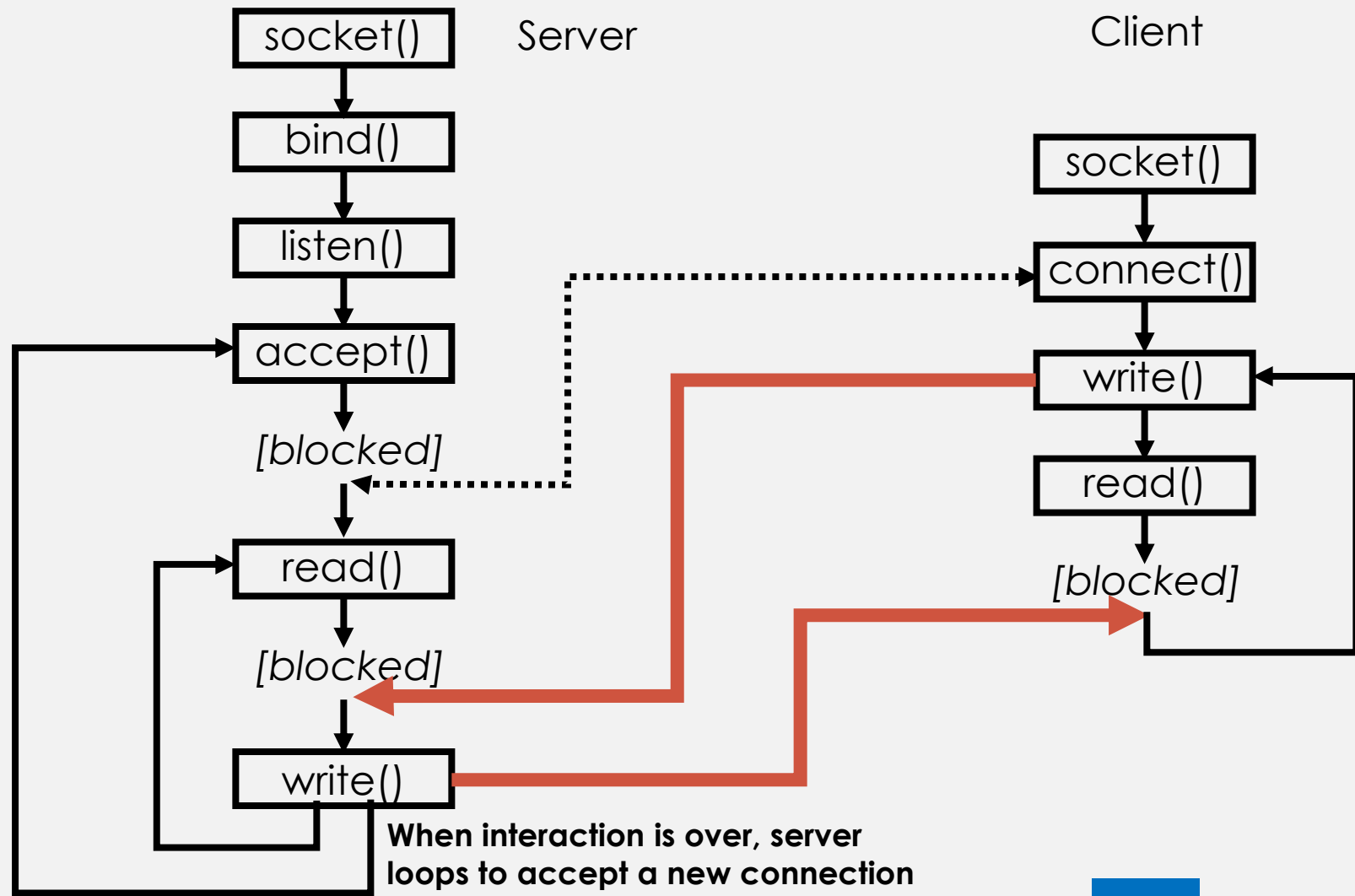
```
def bucket(port=PORT, logfile=None):  
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)  
    s.bind(('', port))  
    print 'waiting on port: %s' % port  
    while 1:  
        try:  
            data, addr = s.recvfrom(BUFSIZE)  
            print `data`[1:-1]  
        except socket.error, msg:  
            print msg
```

- An infinite loop, printing out received messages

EXERCISE 2: MR CREOSOTE DEMO

- This module includes both client and server functionality in a single module
 - `creosote.py`
- Very simple module with no real attempt to use object-oriented features

CONNECTION-ORIENTED SERVICES



TCP

CONNECTION-ORIENTED SERVER

- The socket module
 - Provides access to low-level network programming functions.
 - Example: A server that returns the current time

Time server program

```
from socket import *
import time
s = socket(AF_INET, SOCK_STREAM)
s.bind(("", 8888))
s.listen(5)
while 1:
    client, addr = s.accept()
    print "Got a connection from ", addr
    client.send(time.ctime(time.time()))
    client.close()
```

Create TCP socket
Bind to port 8888
Start listening

Wait for a connection

Send time back



- Notes:
 - Socket first opened by server is not the same one used to exchange data.
 - Instead, the accept() function returns a new socket for this ('client' above).
 - listen() specifies max number of pending connections.

CONNECTION-ORIENTED CLIENT

- Client Program
 - Connect to time server and get current time

Time client program

```
from socket import *
```

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

```
s.connect(("127.0.0.1", 8888))
```

```
tm = s.recv(1024)
```

```
s.close()
```

```
print "The time is", tm
```

Create TCP socket

Connect to server

Receive up to 1024 bytes

Close connection



- Key Points
 - Once connection is established, server/client communicate using send() and recv().
 - Aside from connection process, it's relatively straightforward.
 - Of course, the devil is in the details.
 - And are there ever a LOT of details.

SOCKET FAMILY AND TYPE

- The `socket(family, type, proto)` function
 - Creates a new socket object.
 - family is usually set to `AF_INET`
 - type is one of:
 - **SOCK_STREAM** Stream socket (TCP)
 - **SOCK_DGRAM** Datagram socket (UDP)
- `SOCK_RAW` Raw socket
 - proto is usually only used with raw sockets
 - **IPPROTO_ICMP**
 - **IPPROTO_IP**
 - **IPPROTO_RAW**
 - **IPPROTO_TCP**
 - **IPPROTO_UDP**
- Comments
 - Currently no support for IPv6 (although its on the way).
 - Raw sockets only available to processes running as root.

SOCKET METHODS

- socket methods

• <code>s.accept()</code>	# Accept a new connection
• <code>s.bind(address)</code>	# Bind to an address and port
• <code>s.close()</code>	# Close the socket
• <code>s.connect(address)</code>	# Connect to remote socket
• <code>s.fileno()</code>	# Return integer file descriptor
• <code>s.getpeername()</code>	# Get name of remote machine
• <code>s.getsockname()</code>	# Get socket address as (ipaddr,port)
• <code>s.getsockopt(...)</code>	# Get socket options
• <code>s.listen(backlog)</code>	# Start listening for connections
• <code>s.makefile(mode)</code>	# Turn socket into a file object
• <code>s.recv(bufsize)</code>	# Receive data
• <code>s.recvfrom(bufsize)</code>	# Receive data (UDP)
• <code>s.send(string)</code>	# Send data
• <code>s.sendto(string, address)</code>	# Send packet (UDP)
• <code>s.setblocking(flag)</code>	# Set blocking or nonblocking mode
• <code>s.setsockopt(...)</code>	# Set socket options
• <code>s.shutdown(how)</code>	# Shutdown one or both halves of connection

- Comments

- There are a huge variety of configuration/connection options.
- You'll definitely want a good reference at your side.

UTILITY FUNCTIONS

- This is used for all low-level networking
 - Creation and manipulation of sockets
 - General purpose network functions (hostnames, data conversion, etc...)
 - A direct translation of the BSD socket interface.
- Utility Functions

<code>socket.ntohl(x)</code>	# Convert 32-bit integer to host
<code>socket.ntohs(x)</code>	# Convert 16-bit integer to host order
<code>socket.htonl(x)</code>	# Convert 32-bit integer to network order
<code>socket.htons(x)</code>	# Convert 16-bit integer to network order
<code>socket.inet_aton(ipstr)</code>	# Convert addresses between dotted-quad string
	# format to 32-bit packed binary format
<code>socket.inet_ntoa(packed)</code>	# Convert a 32-bit packed IPv4 address (a string
	# four characters in length) to its standard dotted-
	# quad string representation

- Comments
 - Network order for integers is big-endian.
 - Host order may be little-endian or big-endian (depends on the machine).

HANDLING NAMES & ADDRESSES

- **getfqdn(host= ' ')**
 - Get canonical host name for host
- **gethostbyaddr(ipaddr)**
 - Returns (hostname, aliases, addresses)
 - Hostname is canonical name
 - Aliases is a list of other names
 - Addresses is a list of IP address strings
- **gethostbyname_ex(hostname)**
 - Returns same values as **gethostbyaddr()**

TREATING SOCKETS AS FILES

- **`makefile([mode[, bufsize]])`**
 - Creates a file object that references the socket
 - Makes it easier to program to handle data streams
 - No need to assemble stream from buffers

EXERCISE 3: TCP CLIENT/SERVER

- Run the sample client and server I have provided
 - `tcpserv1.py`
 - `tcpcli1.py`
- Additional questions:
 - What happens if the client aborts (try entering CTRL/D as input, for example)?
 - Can you run two clients against the same server?



SUMMARY OF ADDRESS FAMILIES

- **socket.AF_UNIX**
 - Unix named pipe (NOT Windows...)
- **socket.AF_INET**
 - Internet – IP version 4
 - The basis of this class
- **socket.AF_INET6**
 - Internet – IP version 6
 - Rather more complicated ...

SUMMARY OF SOCKET TYPES

- **socket.SOCK_STREAM**
 - TCP, connection-oriented
- **socket.SOCK_DGRAM**
 - UDP, connectionless
- **socket.SOCK_RAW**
 - Gives access to subnetwork layer
- **SOCK_RDM, SOCK_SEQPACKET**
 - Very rarely used

OTHER SOCKET.* CONSTANTS

- The usual suspects
 - Most constants from Unix C support **SO_***, **MSG_***, **IP_*** and so on
- Most are rarely needed
 - C library documentation should be your guide

TIMEOUT CAPABILITIES

- Originally provided by 3rd-party module
 - Now (Python 2.3) integrated with socket module
- Can set a default for all sockets
 - **`socket.setdefaulttimeout(seconds)`**
 - Argument is float # of seconds
 - Or **`None`** (indicates no timeout)
- Can set a timeout on an existing socket **`s`**
 - **`s.settimeout(seconds)`**

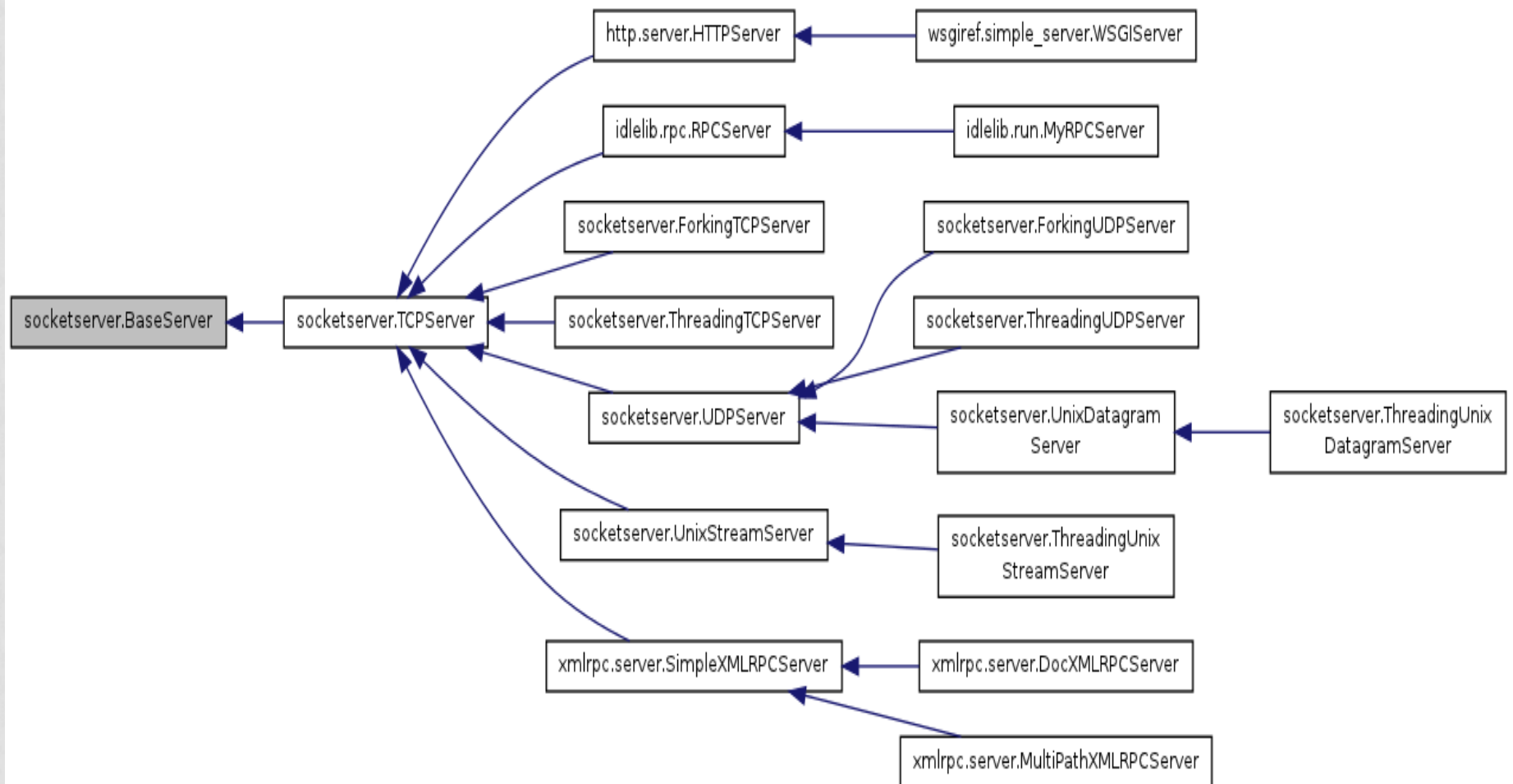
*THE SOCKETSERVER MODULE

- SocketServer is a higher-level module in the standard library
 - (renamed as socketserver in Python 3.x).
- Its goal is to simplify a lot of the boilerplate code that is necessary to create networked clients and servers.
- In this module there are various classes created on your behalf.
- In addition to hiding implementation details from you, another difference is that now we will be writing our applications using classes.
 - Doing things in an object-oriented way helps us organize our data and logically direct functionality to the right places.
 - Our applications will be event-driven
 - they only work when reacting to an occurrence of an event in our system (including the sending and receiving of messages)

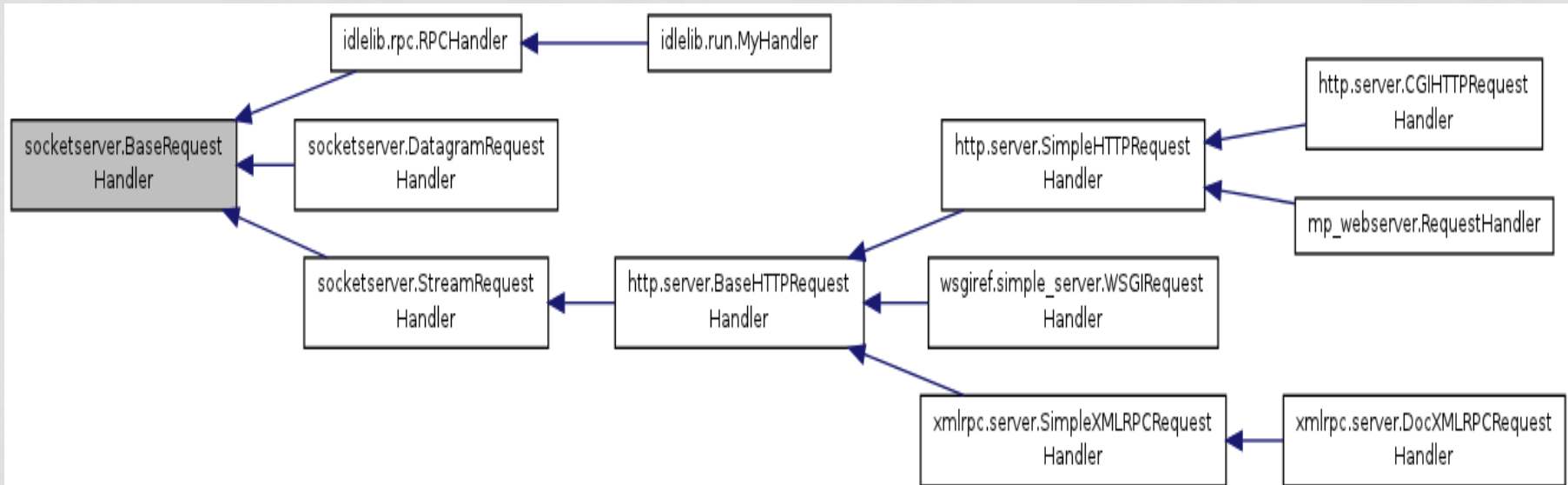
SOCKETSERVER MODULE CLASSES

Class	Description
BaseServer	Contains core server functionality and hooks for mix-in classes; used only for derivation so you will not create instances of this class; use TCPServer or UDPServer instead
TCPServer/ UDPServer	Basic networked synchronous TCP/UDP server
UnixStreamServer/ UnixDatagramServer	Basic file-based synchronous TCP/UDP server
ForkingMixIn/Threading MixIn	Core forking or threading functionality; used only as mix-in classes with one of the server classes to achieve some asynchronicity; you will not instantiate this class directly
ForkingTCPServer/ ForkingUDPServer	Combination of ForkingMixIn and TCPServer/UDPServer
ThreadingTCPServer/ ThreadingUDPServer	Combination of ThreadingMixIn and TCPServer/UDPServer
BaseRequestHandler	Contains core functionality for handling service requests; used only for derivation so you will create instances of this class; use StreamRequestHandler or DatagramRequestHandler instead
StreamRequestHandler/ DatagramRequestHandler	Implement service handler for TCP/UDP servers

INHERITANCE DIAGRAM FOR BASE SERVER



INHERITANCE DIAGRAM FOR BASE REQUEST HANDLER



THE SOCKETSERVER MODULE

- To create a network service, need to inherit from **both** a protocol and handler class.

```
import SocketServer
import time

# This class actually implements the server functionality
class MyRequestHandler(SocketServer.StreamRequestHandler):
    def handle(self):
        print '...connected from:', self.client_address
        self.wfile.write('[%s] %s' % (ctime(),
        self.rfile.readline()))
# Create the server
server = SocketServer.TCPServer(("",21567), MyRequestHandler)
print 'waiting for connection...'
server.serve_forever()
```



tsTservSS.py

- Comments
 - We derive MyRequestHandler as a subclass of SocketServer's StreamRequestHandler and override its handle() method, which is called when an incoming message is received from a client.

SOCKETSERVER TIMESTAMP TCP CLIENT

- The StreamRequestHandler class treats input and output sockets as file-like objects, so we will use readline() to get the client message and write() to send a string back to the client.

```
from socket import *

BUFSIZ = 1024

while True:
    tcpCliSock = socket(AF_INET, SOCK_STREAM)
    tcpCliSock.connect(('localhost', 21567))
    data = raw_input('> ')
    if not data:
        break
    tcpCliSock.send('%s\r\n' % data)
    data = tcpCliSock.recv(BUFSIZ)
    if not data:
        break
    print data.strip()
    tcpCliSock.close()
```



tsTclntSS.py

SERVER LIBRARIES

- **SocketServer** module provides basic server features
- Subclass the **TCPServer** and **UDPServer** classes to serve specific protocols
- Subclass **BaseRequestHandler**, overriding its `handle()` method, to handle requests
- Mix-in classes allow asynchronous handling

USING SOCKETSERVER MODULE

- Server instance created with address and handler-class as arguments:
SocketServer.UDPServer(myaddr, MyHandler)
- Each connection/transmission creates a request handler instance by calling the handler-class*
- Created handler instance handles a message (UDP) or a complete client session (TCP)

* In Python you instantiate a class by calling it like a function

WRITING A HANDLE () METHOD

- **self.request** gives client access
 - **(string, socket)** for UDP servers
 - Connected socket for TCP servers
- **self.client_address** is remote address
- **self.server** is server instance
- TCP servers should handle a complete client session

SKELETON HANDLER EXAMPLES

- No error checking
- Unsophisticated session handling (TCP)
- Simple tailored clients
 - Try telnet with TCP server!
- Demonstrate the power of the Python network libraries

UDP UPPER-CASE SOCKETSERVER

```
# udps1.py
import SocketServer
class UCHandler(SocketServer.BaseRequestHandler):
    def handle(self):
        remote = self.client_address
        data, skt = self.request
        print data
        skt.sendto(data.upper(), remote)
myaddr = ('127.0.0.1', 2345)
myserver = SocketServer.UDPServer(myaddr, UCHandler)
myserver.serve_forever()
```


*Change this function to
alter server's functionality*

- Note: this server never terminates!

UDP UPPER-CASE CLIENT

```
# udpc1.py
from socket import socket, AF_INET, SOCK_DGRAM
srvaddr = ('127.0.0.1', 2345)
data = raw_input("Send: ")
s = socket(AF_INET, SOCK_DGRAM)
s.bind(('', 0))
s.sendto(data, srvaddr)
data, addr = s.recvfrom(1024)
print "Recv:", data
```

Hangs if no response

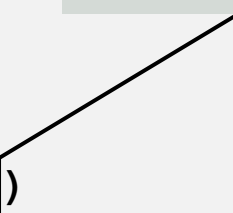


- Client interacts once then terminates

TCP UPPER-CASE SOCKETSERVER

```
# tcps1.py
import SocketServer
class UCHandler(SocketServer.BaseRequestHandler):
    def handle(self):
        print "Connected:", self.client_address
        while 1:
            data = self.request.recv(1024)
            if data == "\r\n":
                break
            print data[:-2]
            self.request.send(data.upper())
myaddr = ('127.0.0.1', 2345)
myserver = SocketServer.TCPServer(myaddr, UCHandler)
myserver.serve_forever()
```

*Change this function to
alter server's functionality*



TCP UPPER-CASE CLIENT

```
# tcpcl.py
from socket import socket, AF_INET, SOCK_STREAM
srvaddr = ('127.0.0.1', 2345)
s = socket(AF_INET, SOCK_STREAM)
s.connect(srvaddr)
while 1:
    data = raw_input("Send: ")
    s.send(data + "\r\n")
    if data == "":
        break
    data = s.recv(1024)
    print data[:-2] # Avoids doubling-up the newline
s.close()
```

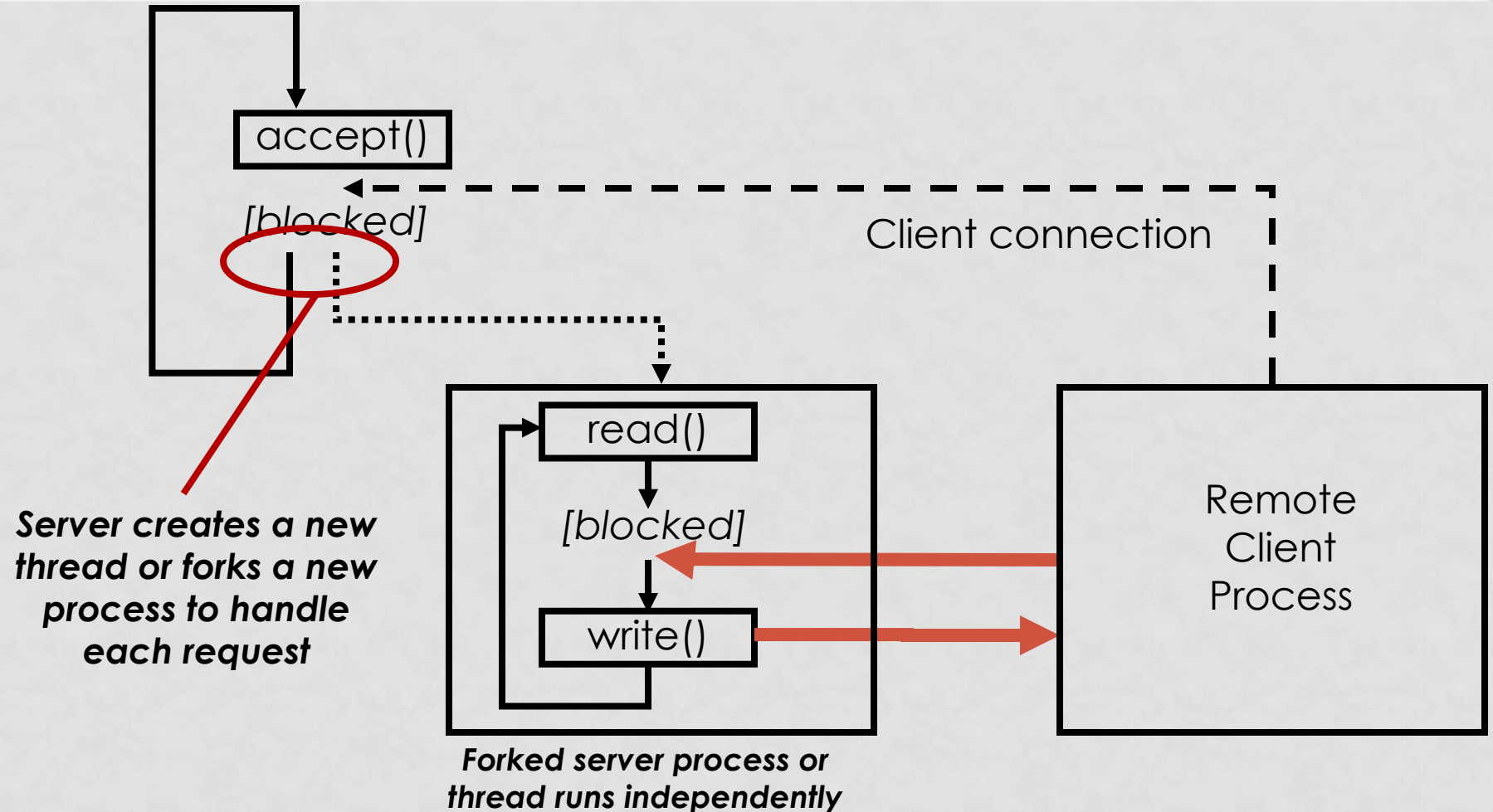

EXERCISE 4: SOCKETSERVER USAGE

- Run the TCP and UDP SocketServer-based servers with the same clients you used before
 - `SockServUDP.py`
 - `SockServTCP.py`
- Additional questions:
 - Is the functionality any different?
 - What advantages are there over writing a "classical" server?
 - Can the TCP server accept multiple connections?

SKELETON SERVER LIMITATIONS (1)

- UDP server adequate for short requests
 - If service is extended, other clients must wait
- TCP server cannot handle concurrent sessions
 - Transport layer queues max 5 connections
 - After that requests are refused
- Solutions?
 - Fork a process to handle requests, or
 - Start a thread to handle requests

SOCKETSERVER FORKING/THREADING



ASYNCHRONOUS SERVER CLASSES

- Use provided asynchronous classes

```
myserver = SocketServer.TCPServer(  
                                myaddr, UHandler)
```

- becomes

```
myserver = SocketServer.ThreadingTCPServer(  
                                myaddr, UHandler)
```

or

```
myserver = SocketServer.ForkingTCPServer(  
                                myaddr, UHandler)
```

IMPLEMENTATION DETAILS

- This is the implementation of all four servers (from `SocketServer.py`):

```
class ForkingUDPServer(ForkingMixIn,  
                       UDPServer): pass
```

```
class ForkingTCPServer(ForkingMixIn,  
                       TCPServer): pass
```

```
class ThreadingUDPServer(ThreadingMixIn,  
                         UDPServer): pass
```

```
class ThreadingTCPServer(ThreadingMixIn,  
                         TCPServer): pass
```

- Uses Python's multiple inheritance
 - Overrides `process_request()` method

MORE GENERAL ASYNCHRONY

- See the **asyncore** and **asynchat** modules
- Use non-blocking sockets
- Based on select using an event-driven model
 - Events occur at state transitions on underlying socket
- Set up a listening socket
- Add connected sockets on creation

EXERCISE 5: ASYNC TCP SERVERS

- Can also be used with UDP, but less often required (UDP often message-response)
 - `SocketServTCPThread.py`
- Very simple to replace threading with forking
 - Non-portable, since forking not supported under Windows (like you care ... ☺)