• 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 ...

Diagram:

Upper layer

| PROTOCOL DATA |

Lower layer

| HDR | DATA |
THE TCP/IP LAYERING MODEL

The Hourglass Model

Applications
UDP  TCP

Data Link

Physical

The Hourglass Model
Example application

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

Network’s domain name: seeecs.edu.pk

Let’s assume the seeecs.edu.pk LAN which allows 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](http://pc.seecs.edu.pk)) to the webserver (the machine [www.seecs.edu.pk](http://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.)

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)
Tell me the IP address of www.seecs.edu.pk?
The IP address of www.seecs.edu.pk is 202.125.157.196
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:

<table>
<thead>
<tr>
<th>Source Port</th>
<th>Destination Port</th>
<th>GET / HTML/1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 1024</td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>
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:**

<table>
<thead>
<tr>
<th>Source IP</th>
<th>Destination IP</th>
<th>Source Port</th>
<th>Destination Port</th>
<th>GET / HTML/1.1</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>Source IP</th>
<th>Destination IP</th>
<th>Source Port</th>
<th>Destination Port</th>
<th>GET / HTML/1.1</th>
</tr>
</thead>
</table>

MAC frame

<table>
<thead>
<tr>
<th>Source MAC address</th>
<th>Destination MAC address</th>
<th>Payload</th>
<th>FCS</th>
</tr>
</thead>
</table>
Encapsulation

Application data → GET / HTML/1.1

TCP Segment

Source Port | Destination Port | Payload
> 1024       | 80               |

IP Packet

Source IP | Destination IP | Payload

MAC Frame

Source MAC address | Destination MAC address | Payload | FCS
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

The index.html page in the wwwroot directory configured for the www.seecs.edu.pk webserver is sent back to the browser for display.

Night view of NUST SEECs Building

Highlights

SEECs Corporate Advisory Council

EXYLENT

Event Calendar

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

Library | Career | Directory | Site Map | E-Mail
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

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

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

<table>
<thead>
<tr>
<th>Telnet</th>
<th>SSH</th>
<th>SMTP</th>
<th>FTP</th>
<th>NFS</th>
<th>DNS</th>
<th>SNMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UDP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IP


Application
- Host-to-host
- Internetwork
- Subnetwork
## COMPARISON BETWEEN STREAMING AND DOWNLOADING

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

• Both protocols (UDP, PCP) 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
  - Ports above 1024 are reserved for user processes.

<table>
<thead>
<tr>
<th>Service</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>FTP</td>
<td>20</td>
</tr>
<tr>
<td>Telnet</td>
<td>23</td>
</tr>
<tr>
<td>SMTP (Mail)</td>
<td>25</td>
</tr>
<tr>
<td>HTTP (WWW)</td>
<td>80</td>
</tr>
<tr>
<td>SSH</td>
<td>22</td>
</tr>
<tr>
<td>DNS</td>
<td>53</td>
</tr>
</tbody>
</table>


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

General
- Socket call
- Write call
- Read call
- Close calls

Server
- Bind call
- Listen call
- Accept call

Client
- Connect call
MAJOR SYSTEM CALLS

Client

socket()
bind()
connect()
send()
recv()
close()

Server

response

socket()
bind()
listen()
accept()

Connection establishment

request

Data request

recv()

data response

End-of-file notification

recv()

send()

close()
<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOCKET</td>
<td>Create a new communication end point</td>
</tr>
<tr>
<td>BIND</td>
<td>Attach a local address to a socket</td>
</tr>
<tr>
<td>LISTEN</td>
<td>Announce willingness to accept connections; give queue size</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>Block the caller until a connection attempt arrives</td>
</tr>
<tr>
<td>CONNECT</td>
<td>Actively attempt to establish a connection</td>
</tr>
<tr>
<td>SEND</td>
<td>Send some data over the connection</td>
</tr>
<tr>
<td>RECEIVE</td>
<td>Receive some data from the connection</td>
</tr>
<tr>
<td>CLOSE</td>
<td>Release the connection</td>
</tr>
</tbody>
</table>
CONNECTIONLESS SERVICES

SERVER

socket()
bind()
recvfrom()
[blocked]
sendto()

CLIENT

socket()
bind()
sendto()
recvfrom()
[blocked]

UDP
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
```
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
  • udpccli1.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
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:]
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

When interaction is over, server loops to accept a new connection

TCP
CONNECTION-ORIENTED SERVER

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

```python
# Time server program
from socket import *
import time
s = socket(AF_INET, SOCK_STREAM)
# Create TCP socket
s.bind(('',8888))
# Bind to port 8888
s.listen(5)
# Start listening
while 1:
    client,addr = s.accept()
    # Wait for a connection
    print "Got a connection from ", addr
    client.send(time.ctime(time.time()))
    # Send time back
    client.close()
```

- 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

```python
# Time client program
from socket import *
s = socket(AF_INET,SOCK_STREAM)  # Create TCP socket
s.connect(("127.0.0.1",8888))  # Connect to server
tm = s.recv(1024)  # Receive up to 1024 bytes
s.close()  # Close connection
print "The time is", tm
```

- 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.

ex2.py
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)
- proto is usually only used with raw sockets
  - IPPROTO_ICMP
  - IPPROTO_IP
  - IPPROTO_RAW
  - IPPROTO_TCP
  - IPPROTO_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
  - `s.accept()` # Accept a new connection
  - `s.bind(address)` # Bind to an address and port
  - `s.close()` # Close the socket
  - `s.connect(address)` # Connect to remote socket
  - `s.fileno()` # Return integer file descriptor
  - `s.getpeername()` # Get name of remote machine
  - `s.getsockname()` # Get socket address as (ipaddr, port)
  - `s.getsockopt(...)` # Get socket options
  - `s.listen(backlog)` # Start listening for connections
  - `s.makefile(mode)` # Turn socket into a file object
  - `s.recv(bufsize)` # Receive data
  - `s.recvfrom(bufsize)` # Receive data (UDP)
  - `s.send(string)` # Send data
  - `s.sendto(string, address)` # Send packet (UDP)
  - `s.setblocking(flag)` # Set blocking or nonblocking mode
  - `s.setsockopt(...)` # Set socket options
  - `s.shutdown(how)` # 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

  ```
  socket.ntohl(x)     # Convert 32-bit integer to host
  socket.ntohs(x)     # Convert 16-bit integer to host order
  socket.htonl(x)     # Convert 32-bit integer to network order
  socket.htons(x)     # Convert 16-bit integer to network order
  socket.inet_aton(ipstr) # Convert addresses between dotted-quad string
                          # format to 32-bit packed binary format
  socket.inet_ntoa(packed) # 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
  • tcperv1.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

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseServer</td>
<td>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</td>
</tr>
<tr>
<td>TCPServer/UDPServer</td>
<td>Basic networked synchronous TCP/UDP server</td>
</tr>
<tr>
<td>UnixStreamServer/UnixDatagramServer</td>
<td>Basic file-based synchronous TCP/UDP server</td>
</tr>
<tr>
<td>ForkingMixIn/Threading MixIn</td>
<td>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</td>
</tr>
<tr>
<td>ForkingTCPServer/ForkingUDPServer</td>
<td>Combination of ForkingMixIn and TCPServer/UDPServer</td>
</tr>
<tr>
<td>ThreadingTCPServer/ThreadingUDPServer</td>
<td>Combination of ThreadingMixIn and TCPServer/UDPServer</td>
</tr>
<tr>
<td>BaseRequestHandler</td>
<td>Contains core functionality for handling service requests; used only for derivation so you will create instances of this class; use StreamRequestHandler or DatagramRequestHandler instead</td>
</tr>
<tr>
<td>StreamRequestHandler/ DatagramRequestHandler</td>
<td>Implement service handler for TCP/UDP servers</td>
</tr>
</tbody>
</table>
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.

```python
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()
```

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.
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.

```python
from socket import *

BUFSIZ = 1024

while True:
    tcpCliSock = socket(AF_INET, SOCK_STREAM)
    tcpCliSock.connect(('localhost', 21567))
    data = raw_input("> ")
    if not data:
        break
data = tcpCliSock.recv(BUFSIZ)
    if not data:
        break
    print data.strip()
tcpCliSock.close()
```
SERVER LIBRARIES

- **SocketServer** module provides basic server features
- Subclass the **TCP`Server** and **UDP`Server** 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:
  ```python
  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()

• Note: this server never terminates!
UDP UPPER-CASE CLIENT

```python
# 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
```

- Client interacts once then terminates

Hangs if no response
TCP UPPER-CASE SOCKETSERVER

```python
# 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
# 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

Server creates a new thread or forks a new process to handle each request.

Forked server process or thread runs independently.

Remote Client Process

Client connection

accept()

[blocked]

read()

[blocked]

write()
• Use provided asynchronous classes

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

• becomes

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

or

```
myserver = SocketServer.ForkingTCPServer(
    myaddr, UCHandler)
```
IMPLEMENTATION DETAILS

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

  ```python
  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)
  • SockServTCPThread.py

• Very simple to replace threading with forking
  • Non-portable, since forking not supported under Windows (like you care ... 😊)