Lecture 13
Networking
Internet architecture

- connects independent heterogeneous networks
  - each network connects multiple computers
  - nearby computers connected by local area network
    often Ethernet but lots of other choices
- networks connected by gateways/routers
  - route packets from one network to next
  - gateways continuously exchange routing information
- each packet passes through multiple gateways
  - gateway passes packet to gateway that is closer to ultimate destination
  - usually operated by different companies
- information travels through networks in packets
  - each packet is independent of all others
    like individual envelopes through the mail
  - all packets have the same format
    but are carried on different physical transport media
- no central control
- ICANN: central authority for resources that have to be unique
  - IP addresses, domain names, country codes, ...
Protocols

• precise rules that govern communication between two parties
• basic Internet protocols usually called TCP/IP
  – 1973 by Bob Kahn *64, Vint Cerf
• **IP: Internet protocol (bottom level)**
  – all packets shipped from network to network as IP packets
  – each physical network has own format for carrying IP packets (Ethernet, fiber, …)
  – no guarantees on quality of service or reliability: "best effort"
• **TCP: transmission control protocol**
  – reliable stream (circuit) transmission in 2 directions
  – most things we think of as "Internet" use TCP
• **application-level protocols, mostly built from TCP**
  – SSH, FTP, SMTP (mail), HTTP (web), …
• **UDP: user datagram protocol**
  – unreliable but simple, efficient datagram protocol
  – used for DNS, NFS, …
• **ICMP: internet control message protocol**
  – error and information messages
  – ping, traceroute
Internet (IP) addresses

• each network and each connected computer has an IP address

• IP address: a unique 32-bit number in IPv4 (IPv6 is 128 bits)
  – 1st part is network id, assigned centrally in blocks
    (Internet Assigned Numbers Authority -> Internet Service Provider -> you)
  – 2nd part is host id within that network
    assigned locally, often dynamically

<table>
<thead>
<tr>
<th>net part</th>
<th>host on that net</th>
</tr>
</thead>
</table>

• written in "dotted decimal" notation: each byte in decimal
  – e.g., 128.112.128.81 = www.princeton.edu

| 128 | 112 | 128 | 81 |

| 10000000 | 01110000 | 10000000 | 01010001 |
IPv6

An IPv6 address (in hexadecimal)

```
2001:0DB8:AC10:FE01:0000:0000:0000:0000
```

Zeroes can be omitted

```
2001:0DB8:AC10:FE01::
```

```
0010000000000000:0000:0000:0000:0000:0000:0000:0000
```

Fixed header format

<table>
<thead>
<tr>
<th>Offsets</th>
<th>Octet</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Octet</strong></td>
<td><strong>Bit</strong></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td><strong>Version</strong></td>
<td><strong>Traffic Class</strong></td>
<td><strong>Flow Label</strong></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td><strong>Payload Length</strong></td>
<td><strong>Next Header</strong></td>
<td><strong>Hop Limit</strong></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>95</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>160</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>192</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>224</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>288</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
IP: Internet Protocol

- **unreliable connectionless packet delivery service**
  - every packet has 20-40B header with
    source & destination addresses,
    time to live: maximum number of hops before packet is discarded (each gateway decreases this by 1)
    checksum of header information (not of data itself)
  - up to 65 KB of actual data

- **IP packets are datagrams:**
  - individually addressed packages, like envelopes in mail
  - "connectionless": every packet is independent of all others
  - unreliable -- packets can be damaged, lost, duplicated, delivered out of order
  - packets can arrive too fast to be processed
  - stateless: no memory from one packet to next
  - limited size: long messages have to be fragmented and reassembled

- **higher level protocols synthesize error-free communication from IP packets**
TCP: Transmission Control Protocol

- reliable connection-oriented 2-way byte stream
  - no record boundaries
    - if needed, create your own by agreement
- a message is broken into 1 or more packets
- each TCP packet has a header (20 bytes) + data
  - header includes checksum for error detection,
  - sequence number for preserving proper order, detecting missing or duplicates
- each TCP packet is wrapped in an IP packet
  - has to be positively acknowledged to ensure that it arrived safely
    - otherwise, re-send it after a time interval
- a TCP connection is established to a specific host
  - and a specific "port" at that host
- each port provides a specific service
  - see /etc/services
    - FTP = 21, SSH = 22, SMTP = 25, HTTP = 80
- TCP is basis of most higher-level protocols
Higher level protocols:

- FTP: file transfer
- SSH: terminal session
- SMTP: mail transfer
- HTTP: hypertext transfer -> Web
- protocol layering:
  - a single protocol can't do everything
  - higher-level protocols build elaborate operations out of simpler ones
  - each layer uses only the services of the one directly below
  - and provides the services expected by the layer above
  - all communication is between peer levels: layer N destination receives exactly the object sent by layer N source
Internet Exchange Point (IXP)

NYIIX Architecture

The NYIIX infrastructure consists of five nodes to cover three major TELCO buildings, 85 10th Avenue, 7 Teleport Drive, 25 Broadway, 60 Hudson Street, 111 8th Avenue and 32 Avenue of Americas in New York City. All five nodes are connected and any member can peer with any member regardless of its location.

The NYIIX address space is 198.32.160.0/24 for IPv4 peering and 2001:0504:0001::/64 for IPv6 peering. A member can have either an IPv4 address, an IPv6 address, or both.

The NYIIX supports any type of connections:
- 10-100 BASE-T Half/Full Duplex
- 1000 BASE-SX via MMF
- 1000 BASE-LX via SMF
- 10GigE LR 1310nm via SMF
Network programming

- C: client, server, socket functions; based on processes & inetd
- Java: import java.net.* for Socket, ServerSocket; threads
- Python: import socket, SocketServer; threads
- underlying mechanism (pseudo-code):
  
  server:
  ```
  fd = socket(protocol)
  bind(fd, port)
  listen(fd)
  fd2 = accept(fd, port)
  while (...)
      read(fd2, buf, len)
      write(fd2, buf, len)
  close(fd2)
  ```

  client:
  ```
  fd = socket(protocol)
  connect(fd, server IP address, port)
  while (...)
      write(fd, buf, len)
      read(fd, buf, len)
  close(fd)
  ```
C TCP client

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

struct hostent *ptrh; /* host table entry */
struct protoent *ptrp; /* protocol table entry */
struct sockaddr_in sad; /* server adr */
sad.sin_family = AF_INET; /* internet */
sad.sin_port = htons((u_short) port);
ptrh = gethostbyname(host); /* IP address of server /
memcpy(&sad.sin_addr, ptrh->h_addr, ptrh->h_length);
ptrp = getprotobynamel("tcp");
fd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
connect(sd, (struct sockaddr *) &sad, sizeof(sad));

while (...) {
    write(fd, buf, strlen(buf)); /* write to server */
    n = read(fd, buf, N); /* read reply from server */
}
close(fd);
C TCP server

struct protoent *ptrp;  /* protocol table entry */
struct sockaddr_in sad;  /* server adr */
struct sockaddr_in cad;  /* client adr */
memset((char *) &sad, 0, sizeof(sad));
sad.sin_family = AF_INET;  /* internet */
sad.sin_addr.s_addr = INADDR_ANY; /* local IP adr */
sad.sin_port = htons((u_short) port);
ptrp = getprotobyname("tcp");
fd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
bind(fd, (struct sockaddr *) &sad, sizeof(sad));
listen(fd, QLEN);

while (1) {
    fd2 = accept(sd, (struct sockaddr *) &cad, &alen));
    while (1) {
        read(fd2, buf, N);
        write(fd2, buf, N);
    }
    close(fd2);
}
Java networking classes

• **Socket**
  – client side
  – basic access to host using TCP
    reliable, stream-oriented connection

• **ServerSocket**
  – server side
  – listens for TCP connections on specified port
  – returns a Socket when connection is made

• **DatagramSocket: UDP datagrams**
  – unreliable packet service

• **URL, URLConnection**
  – high level access: maps URL to input stream
  – knows about ports, services, etc.

• **import java.net.***
Client: copy stdin to server, read reply

- uses Socket class for TCP connection between client & server

```java
import java.net.*;
import java.io.*;

public class cli {

    static String host = "localhost";  // or 127.0.0.1
    static String port = "33333";

    public static void main(String[] argv) {
        if (argv.length > 0)
            host = argv[0];
        if (argv.length > 1)
            port = argv[1];
        new cli(host, port);
    }

    (continued...)
```
Client: part 2

cli(String host, String port) { // tcp/ip version
    try {
        BufferedReader stdin = new BufferedReader(
            new InputStreamReader(System.in));
        Socket sock = new Socket(host, Integer.parseInt(port));
        System.err.println("client socket " + sock);
        BufferedReader sin = new BufferedReader(
            new InputStreamReader(sock.getInputStream()));
        BufferedWriter sout = new BufferedWriter(
            new OutputStreamWriter(sock.getOutputStream()));
        String s;
        while ((s = stdin.readLine()) != null) { // read cmd
            sout.write(s); // write to socket
            sout.newLine();
            sout.flush(); // needed
            String r = sin.readLine(); // read reply
            System.out.println(host + " got [" + r + "]");
            if (s.equals("exit"))
                break;
        }
        sock.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
Single-thread Java server

- server: echoes lines from client

```java
public class srv {
    static String port = "33333";
    public static void main(String[] argv) {
        if (argv.length == 0)
            new srv(port);
        else
            new srv(argv[0]);
    }
    srv(String port) {
        // tcp/ip version
        try {
            ServerSocket ss = new ServerSocket(Integer.parseInt(port));
            while (true) {
                Socket sock = ss.accept();
                System.err.println("server socket " + sock);
                new echo(sock);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```
Rest of server

class echo {
    Socket sock;
    echo(Socket sock) throws IOException {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(sock.getInputStream())); // from socket
        BufferedWriter out = new BufferedWriter(
            new OutputStreamWriter(sock.getOutputStream())); // to socket
        String s;
        while ((s = in.readLine()) != null) {
            out.write(s);
            out.newLine();
            out.flush();
            if (s.equals("exit"))
                break;
        }
        sock.close();
    }
}

- this is single-threaded: only serves one client at a time
Serving multiple requests simultaneously

- how can we serve more than one client at a time?
- **in C/Unix, usually start a new process for each conversation**
  - fork & exec: process is entirely separate entity
  - usually shares nothing with other processes
  - operating system manages scheduling
  - alternative: use a threads package (e.g., pthreads)
- **in Java, use threads**
  - threads all run in the same process and address space
  - process itself controls allocation of time (JVM)
  - threads have to cooperate (JVM doesn't enforce this)
  - threads must not interfere with each other's data and use of time
- **Thread class defines two primary methods**
  - start start a new thread
  - run run this thread
- **a class that wants multiple threads must**
  - extend Thread
  - implement run()
  - call start() when ready, e.g., in constructor
**Inetd**

- how do we arrange that a server can dispatch requests to the right processes without blocking?
- one solution: a daemon process that accepts connection requests, and forks a new process for each request

```c
for (;;) {
    int alen = sizeof(cad), sd2;
    if ((sd2 = accept(sd, (struct sockaddr *) &cad, &alen)) < 0) {
        exit(1);      /* accept failed */
    }
    if (fork() == 0) {
        close(sd);    /* child does this */
        runsrv(sd2);
        exit(0);
    }
    close(sd2);    /* parent does this */
}
```
Multi-threaded server

```java
public class multisrv {
    static String port = "33333";

    public static void main(String[] argv) {
        if (argv.length == 0)
            multisrv(port);
        else
            multisrv(argv[0]);
    }

    public static void multisrv(String port) { // tcp/ip version
        try {
            ServerSocket ss =
                new ServerSocket(Integer.parseInt(port));
            while (true) {
                Socket sock = ss.accept();
                System.err.println("multiserver " + sock);
                new echo1(sock);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```
Thread part...

class echo1 extends Thread {

echo1(Socket sock) {
    this.sock = sock; start();
}

public void run() {
    try {
        BufferedReader in = new BufferedReader(new InputStreamReader(sock.getInputStream()));
        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(sock.getOutputStream()));
        String s;
        while ((s = in.readLine()) != null) {
            out.write(s);
            out.newLine();
            out.flush();
            System.err.println(sock.getInetAddress() + " " + s);
            if (s.equals("exit")) // end this conversation
                break;
            if (s.equals("die!")) // kill the server
                System.exit(0);
        }
        sock.close();
    } catch (IOException e) {
        System.err.println("server exception " + e);
    }
}
}
Multi-threaded Python server

#!/usr/bin/python

import SocketServer
import socket
import string

class Srv(SocketServer.StreamRequestHandler):
    def handle(self):
        print "Python server called by %s" % (self.client_address,)
        while 1:
            line = self.rfile.readline()
            print "server got " + line.strip()
            self.wfile.write(line)
            if line.strip() == "exit":
                break

srv = SocketServer.ThreadingTCPServer(('',33333), Srv)
srv.serve_forever()
Node.js server

```javascript
var net = require('net');
var os = require('os');
var server = net.createServer(function(c) {
    // 'connection' listener
    console.log('server connected');
    c.on('data', function(d) {
        process.stdout.write(d);
        console.log("Javascript srv got [%s] from %s",
            d.toString().trim(), os.hostname());
    });
    c.on('end', function() {
        console.log('server disconnected');
    });
    c.pipe(c);
});
server.listen(33333, function() {
    // 'listening' listener
    console.log('Javascript srv listening');
});
```
Multi-threaded client: web crawler

• want to crawl a bunch of web pages to do something
  – e.g., figure out how big they are

• problem: network communication takes relatively long time
  – program does nothing useful while waiting for a response

• solution: access pages in parallel
  – send requests asynchronously
  – display results as they arrive
  – needs some kind of threading or other parallel process mechanism

• takes less time than doing them sequentially
import urllib2, time, sys

def main():
    start = time.time()
    for url in sys.argv[1:]:
        count("http://" + url)
    dt = time.time() - start
    print "\n\ntotal: %.2fs" % (dt)

def count(url):
    start = time.time()
    n = len(urllib2.urlopen(url).read())
    dt = time.time() - start
    print "%6d  %6.2fs  %s" % (n, dt, url)

main()
import urllib2, time, sys, threading

global_lock = threading.Lock()

class Counter(threading.Thread):
    def __init__(self, url):
        super(Counter, self).__init__()
        self.url = url

    def count(self, url):
        start = time.time()
        n = len(urllib2.urlopen(url).read())
        dt = time.time() - start
        with global_lock:
            print "%6d  %6.2fs  %s" % (n, dt, url)

    def run(self):
        self.count(self.url)

def main():
    threads = []
    start = time.time()
    for url in sys.argv[1:]:  # one thread each
        w = Counter("http://" + url)
        threads.append(w)
        w.start()

    for w in threads:
        w.join()
    dt = time.time() - start
    print "\ntotal: %.2fs" % (dt)

main()