Lecture 13
Networking
CAS: Central Authentication Service

• if your project requires users to log in with a Princeton netid
don't ask users to send you their passwords at all, and
especially not in the clear

• OIT provides a central authentication service
  – the user visits your startup page
  – the user is asked to authenticate via OIT's service
  – the name and password are sent to an OIT site for validation
    (without passing through your code at all)
  – if OIT authenticates the user, your code is called

• OIT web page about CAS:
  https://sp.princeton.edu/oit/eis/iam/authentication/
  CAS/CAS%20Developer%20KB.aspx

sample code:
  www.cs.princeton.edu/~bwk/public_html/CAS
Authentication for projects (etc.)

- **PHP version**
  ```php
  <?php
  require 'CASClient.php';
  $C = new CASClient();
  $netid = $C->Authenticate();
  echo "Hello $netid"; // or other code
  ?>
  ```

- **Python version**
  ```python
  import CASClient, os
  C = CASClient.CASClient()
  netid = C.Authenticate()
  print "Content-Type: text/html\n"
  print "Hello %s" % netid # or other code
  ```

- **Java version**
  ```java
  CASClient casClient = new CASClient();
  String netid = casClient.authenticate();
  System.out.println("Content-type: Text/html\n");
  System.out.println("Hello "+ netid);
  ```
Behind the scenes in the client libraries

• your web page sends user to
  https://fed.princeton.edu/cas/login?
    service=url-where-user-will-log-in

• CAS sends user back to the service url to log in
  with a parameter ticket=hash-of-something

• your login code sends this back to
  https://fed.princeton.edu/cas/validate?
    ticket=hash&service=url…log-in

• result from this is either 1 line with "no"
  or two lines with "yes" and netid
Where do we go from here?

• networking
• C++, STL, Java collections
• Go
• Little languages
• Design patterns
• ???

• Guests:
  Apr 6: Bjarne Stroustrup, creator of C++
  Apr 13: Clay Bavor '05, head of virtual reality at Google
  Apr 20: Bonnie Eisenman '14, Codecademy and Twitter
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, ...
Local Area Networks; Ethernet

- a LAN connects computers (“hosts”) in a small geographical area
- Ethernet is the most widely used LAN technology
  - developed by Bob Metcalfe & David Boggs (ELE '72) at Xerox PARC, 1973
  - each host has a unique 48-bit identification number
  - data sent in "packets" of 100-1500 bytes
    packets include source and destination addresses, error checking
    typical data rate 100-1000 Mbits/sec; maximum cable lengths
  - CSMA/CD: carrier sense multiple access with collision detection
    sender broadcasts, but if detects someone else sending, stops, waits a random interval, tries again
  - hubs and wireless nets simulate cable behavior

<table>
<thead>
<tr>
<th>Packet Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>hdr</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

- coaxial cable
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

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

<table>
<thead>
<tr>
<th>net part</th>
<th>host on that net</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>112</td>
</tr>
<tr>
<td>128</td>
<td>81</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10000000</th>
<th>01110000</th>
<th>10000000</th>
<th>01010001</th>
</tr>
</thead>
<tbody>
<tr>
<td>net part</td>
<td>host part</td>
<td>net part</td>
<td>host part</td>
</tr>
</tbody>
</table>
IP

• **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
YOU WANT YOUR COUSIN TO SEND YOU A FILE? EASY. HE CAN EMAIL IT TO— ... OH, IT’S 25 MB? HMM...

DO EITHER OF YOU HAVE AN FTP SERVER? NO, RIGHT.

IF YOU HAD WEB HOSTING, YOU COULD UPLOAD IT...

HMM. WE COULD TRY ONE OF THOSE MEGASHARE UPLOAD SITES, BUT THEY'RE FLAKY AND FULL OF DELAYS AND PORN POPUPS.

HOW ABOUT AIM DIRECT CONNECT? ANYONE STILL USE THAT?

OH, WAIT, DROPBOX! IT'S THIS RECENT STARTUP FROM A FEW YEARS BACK THAT SYNCS FOLDERS BETWEEN COMPUTERS. YOU JUST NEED TO MAKE AN ACCOUNT, INSTALL THE-

OH, HE JUST DROVE OVER TO YOUR HOUSE WITH A USB DRIVE?

UH, COOL, THAT WORKS, TOO.

I LIKE HOW WE'VE HAD THE INTERNET FOR DECADES, YET "SENDING FILES" IS SOMETHING EARLY ADOPTERS ARE STILL FIGURING OUT HOW TO DO.
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
#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 addr */
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 = getprotobynam("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

```c
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.*`
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...)
}
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(port) {
        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();
        }
    }
}
```
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 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
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();
        }
    }
}
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 server = net.createServer(function(c){
  // 'connection' listener
  console.log('server connected');
  c.on('end', function() {
    console.log('server disconnected');
  });
  c.pipe(c);
});
server.listen(33333, function() { // 'listening' listener
  console.log('server bound');
});
```
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 "total: \%.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()
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()
```python
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)
```