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/sdp/CAS/`
  `Wiki%20Pages/Home.aspx`

- **sample code:**

  `www.cs.princeton.edu/~bwk/public_html/CAS`

Authentication for projects (etc.)

- **PHP version**

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

- **Python version**

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

- **Java version**

```
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`*

Source code management systems

- **SVN, Git, Mercurial, Bazaar, Perforce, ...**
- **for managing large projects with multiple people**
  - work locally or across a network
- **store and retrieve all versions of all directories and files in a project**
  - source code, documentation, tests, binaries, ...
- **support multiple concurrent users**
  - independent editing of files
  - merged into single version
- **highly recommended for COS 333 projects!**
  - save all previous versions of all files so you can back out of a bad change
  - log changes to files so you can see who changed what and why
  - maintain consistency by resolving mediate conflicting changes made by different users

Alternatives

- Git

  http://git-scm.com/

- SVN

  http://subversion.apache.org/

- Bazaar

  http://bazaar-vcs.org

- Mercurial

  http://www.selenic.com/mercurial

- Perforce

  http://www.perforce.com

Basic sequence for all systems

- **create a repository that holds copies of your files**
  - including all changes and bookkeeping info
- **each person checks out a copy of the files**
  - "copy - modify - merge" model
  - get files from repository to work on
    does not lock the repository
  - make changes in a local copy
  - when satisfied, check in (== commit) changes
- **if my changes don't conflict with your changes**
  - system updates its copies with the revised versions
  - automatically merges edits on different lines
  - keeps previous copies
- **if my changes conflict with your changes**
  - e.g., we both changed lines in the same part of file,
    checkin is not permitted
  - we have to resolve the conflict manually

Git

- **originally written by Linus Torvalds, 2005**
- **distributed**
  - no central server: every working directory is a complete repository
  - has complete history and revision tracking capabilities
- **originally for maintaining Linux kernel**
  - lots of patches
  - many contributors
  - very distributed
  - dispute with BitKeeper (commercial system)
  - dissatisfaction with CVS / SVN

Basic Git sequences (git-scm.com/documentation, gitref.org)

```
cd project
git init
```

makes .git repository

```
git add .
git commit
```

makes a snapshot of current state

[modify files]

```
git add …  [for new ones]
git rm …   [for dead ones]
git commit
git log --stat –summary
git clone [url]
```

makes a copy of a repository

Basic sequence for SVN

- **do this once:**

```
svnadmin create repository
[mkdir proj.dir & put files in it, or use existing directory ]
svn import proj.dir file:///repository -m 'initial repository'
svn checkout file:///repository working.dir
```

- **create, edit files in working.directory**

```
cd working.dir
ed x.c    # etc.
svn diff x.c
svn add newfile.c
```

- **update the repository from the working directory**

```
svn commit  # commit all the changes
```

Basic sequence, continued

- **when changes are committed, SVN insists on a log message**
  - strong encouragement to record what change was made and why
  - can get a history of changes to one or more files
  - can run diff to see how versions of a file differ

- **can create multiple branches of a project**

- **can tag snapshots for, e.g., releases**

- **can be used as client-server over a network, so can do distributed development**
  - repository on one machine
  - users and their local copies can be anywhere

Networking overview

- **a bit of history**
- **local area networks**
- **Internet**
  - protocols, …
- **network plumbing and software**

Internet mechanisms

- **names** for networks and computers
  - `www.cs.princeton.edu, de.licio.us`
  - hierarchical naming scheme
  - imposes logical structure, not physical or geographical
- **addresses** for identifying networks and computers
  - each has a unique 32-bit IP address (IPv6 is 128 bits)
  - ICANN assigns contiguous blocks of numbers to networks (icann.org)
  - network owner assigns host addresses within network
- **DNS** Domain Name System maps names /addresses
  - `www.princeton.edu = 128.112.136.12`
  - hierarchical distributed database
  - caching for efficiency, redundancy for safety
- **routing to** find paths from network to network
  - gateways/routers exchange routing info with nbrs
- **protocols** for packaging and transporting information, handling errors, ...
  - IP (Internet Protocol): a uniform transport mechanism
  - at IP level, all info is in a common packet format
  - different physical systems carry IP in different formats (e.g., Ethernet, wireless, fiber, phone,...)
  - higher-level protocols built on top of IP for exchanging info like web pages, mail, …

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

net part	**host on that net**

- **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**

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

IP

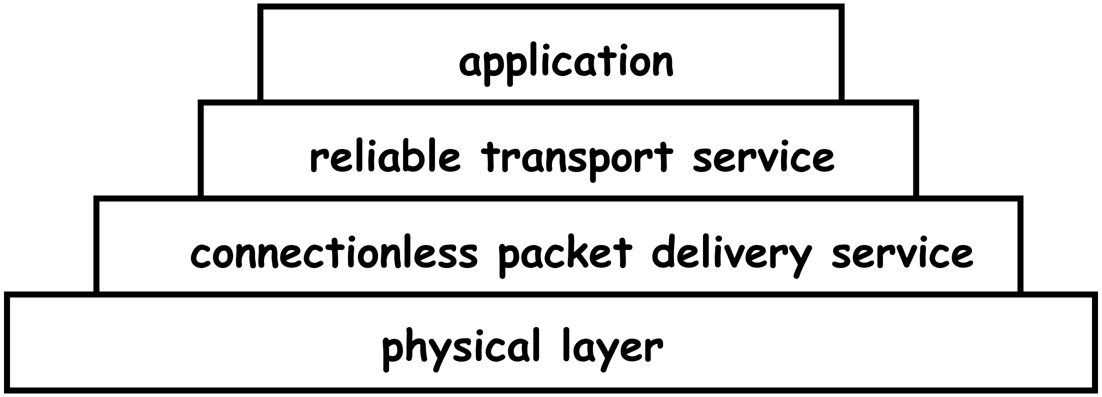
- **unreliable connectionless packet delivery service**
  - every packet has 20-40B header with
    - source & destination addresses,
    - time to live: maximum number of hops before packet is discarded (each gateway decreases this by 1)
    - checksum of header information (not of data itself)
  -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



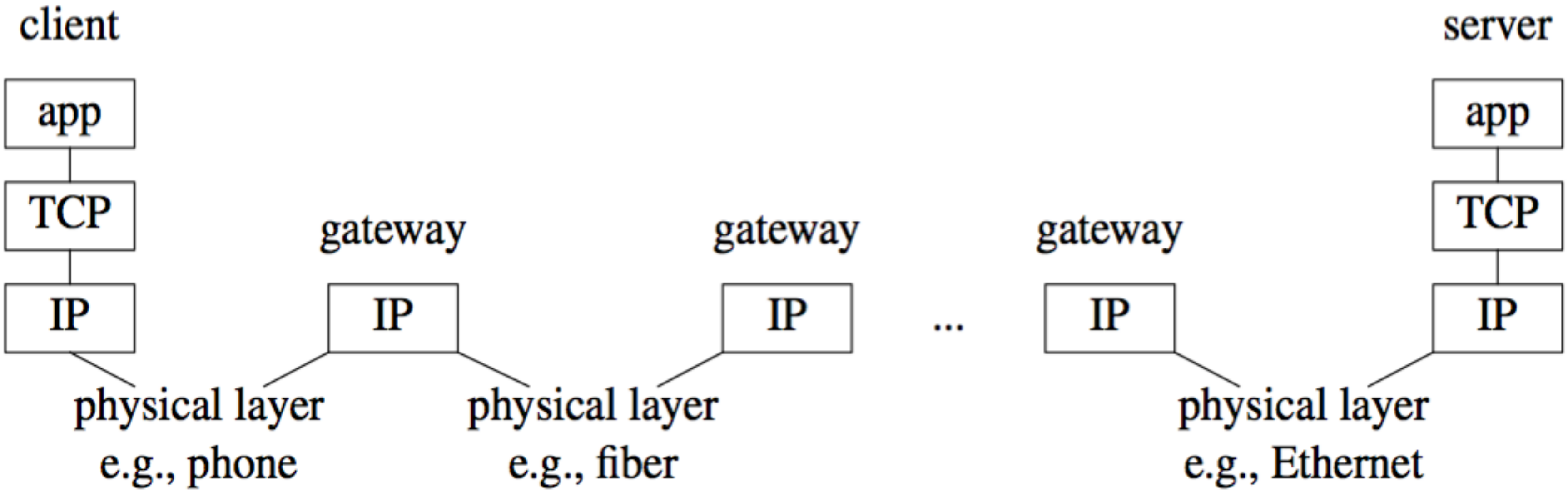
application

reliable transport service

connectionless packet delivery service

physical layer

How information flows



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 = getprotobyname("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**

```
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**

```
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) {    // 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 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

Multi-threaded server

```
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

```
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');
});
```