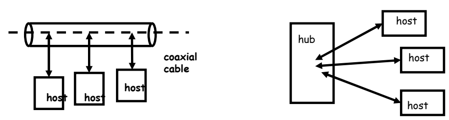
Local Area Networks; Ethernet

- **a LAN connects computers in a small area**
- **Ethernet is the most widely used LAN technology**
  - developed by Bob Metcalfe & David Boggs (Xerox PARC, 1973)
  - each host has a unique 48-bit identification number
  - data sent in "packets" of 100-1500 bytes
    - includes source and destination addresses, error checking
    - data rate 10-1000 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



packet:

hdr	src	dest		data	CRC
8	6	6	2	46-1500	4

Internet

- **independent but connected networks**
  - each network connects multiple computers
  - nearby computers connected by local area network
    - often Ethernet but lots of other choices
- **information travels through networks in packets**
  - each packet independent of all others
    - like individual envelopes through the mail
  - all packets have the same format
- **networks connected by gateway computers (routers)**
  - route packets of information 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

What it needs to work:

- **addresses** for identifying networks and computers
  - each has a unique 32-bit number ("IP address")
  - central authority assigns numbers to networks
    ICANN: Internet Corporation for Assigned Names and Numbers
  - network owner assigns host addresses within network
- **names** for computers
  - **cs.research.bell-labs.com, cs.princeton.edu**
- **mapping** from names to addresses and back
- **routing** for finding paths from network to network
- **protocols** for packaging and transporting information
  - IP or "Internet Protocol": a uniform transport mechanism
  - at IP level, all information is in a common format
  - different physical systems carry IP in different formats
  - higher-level protocols built on top of IP for exchanging information like web pages, mail, …

Domain Name System (DNS)

- **DNS converts names to IP addresses and vice versa**
  - www.princeton.edu == 128.112.128.15
  - www.carnegiehall.org == 65.17.202.130
  - a.root-servers.net == 198.41.0.30
- **hierarchical naming and searching**
  - ICANN controls top level domain names
  - delegates responsibility to levels below for administration and translation into IP addresses
  - each level responsible for names within it
    princeton.edu handles all of Princeton
    delegates cs.princeton.edu to a CS machine
- **top level domains include .com, .edu, .gov, .xx for country XX, etc.**
- **lookup for a name asks a local name server first (nslookup)**
  - if not known locally, ask a server higher up, ...
  - recently-used names are cached to speed up access
- **names impose logical structure, not physical or geographical**
- **names have significant commercial value**

Routing

- **networks are connected by gateways or routers**
- **routing rules direct packets from gateway to gateway**

  trying to get closer to ultimate destination
- **routers exchange information about routes**
- **bottom-up view:**
  - gateways move packets from one network to another based on network id
  - if destination on same network, use physical address
  - otherwise send to a gateway, which passes it to another network
- **top-down view:**
  - networks connected only through gateways
  - core has a small set of gateways that exchange complete routing info about which nets it knows about and number of hops to reach them
  - autonomous system: group of networks under single authority
  - passes reachability info to core for use by other autonomous systems
  - interior gateway protocols exchange routing info within a single AS
- **traceroute: how do you get to Carnegie Hall?**

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 (e.g., 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**
  - Telnet, 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

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

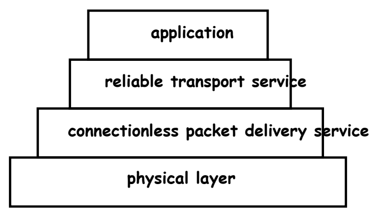
- **reliable connection-oriented 2-way byte stream**
- **a message is broken into 1 or more packets**
- **each TCP packet has a header (20 bytes) + data**
  - header includes checksum for error detection,
  - sequence number for preserving proper order, detecting missing or dups
- **each TCP packet is 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 <u>port</u> provides a specific service**
  - FTP = 21, Telnet = 23, HTTP = 80
    - SMTP 25, daytime 13, echo 7

- **TCP is the most used protocol**
  - basis of almost all higher-level protocols
  - ~15,000 lines of C for TCP/IP

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



Network code

- ***C***
  - client, server, socket functions (similar in Perl)
  - processes & inetd
- **Java**
  - java.net.*
  - Socket class
  - ServerSocket class
  - URL classes

- **underlying mechanism (pseudo-code):**

server:

```
fd = socket(protocol)
bind(fd, port)
listen(fd)
fd2 = accept(fd, port)
    read(fd2, buf, len)
    write(fd2, buf, len)
 close(fd2)
```

client:

```
fd = socket(protocol)
connect(fd, server IP address, port)
   write(fd, buf, len)
   read(fd, buf, len)
close(fd)
```

Client-server in TCP/IP

```
client                          server

                                s = socket()
                                bind(s, port)
                                listen(s)
fd = socket()
connect(fd, srv, port)          s2 = accept(s)

while () {                      while () {
     write fd                       read s2
     read fd                        write s2
}                               }
close fd                        close s2
```

C network client

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.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");

sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
connect(sd, (struct sockaddr *) &sad, sizeof(sad));

write(sd, buf, strlen(buf2));      /* write to server */
n = read(sd, buf, N);    /* read reply from server */

close(sd);
```

C 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");
sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
bind(sd, (struct sockaddr *) &sad, sizeof(sad));
listen(sd, QLEN) ;

   while (1) {
     sd2 = accept(sd, (struct sockaddr *) &cad, &alen));
     while (1) {
        read(sd2, buf, N);
        write(sd2, buf, N);
     }
     close(sd2);
   }
```

Perl client

```
#!/usr/local/bin/perl -w

use strict;
use Socket;

my $host = shift || 'localhost';
my $port = shift || 5194;
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
my $proto = getprotobyname('tcp');

socket(SOCK, PF_INET, SOCK_STREAM, $proto)
        or die "socket: $!";
connect(SOCK, $paddr) or die "connect: $!";

while (<STDIN>) {
  syswrite(SOCK, $_, length($_));
  my $reply = <SOCK>;
  print "reply from srv = [$reply]\n";
}
close(SOCK);
```

Perl client with IO::Socket module

- **Perl module hides underlying calls**

```
#!/usr/local/bin/perl -w

use strict;
use IO::Socket;

my $host = shift || 'localhost';
my $port = shift || 5194;
my $fh = IO::Socket::INET->new("$host:$port")
  or die "connect: $!";

print "starting Perl client calling $host $port\n";

while (<STDIN>) {
  print $fh $_;
  my $reply = <$fh>;
  print "reply from srv = [$reply]\n";
  last if ($reply =~ /exit/);
}
close($fh);
```

Java Internet 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**
  - high level access: maps URL to input stream
  - knows about ports, services, etc.
  - URLConnection class provides more control
- **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 client {

static String host = "localhost";
static String port = "5194";

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

- (continued…)

Client: part 2

```
client(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 + " reply " + r);
            if (s.equals("exit"))
                break;
      }
      sock.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Single-thread Java server

- **server: echoes lines from client**

```
public class server {

static String port = "5194";

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

server(String port) {     // tcp/ip version
    try {
        ServerSocket srv =
            new ServerSocket(Integer.parseInt(port));
        while (true) {
            Socket sock = srv.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 services one client at a time

Serving multiple requests simultaneously

- **how can we run 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)
  - threads have to be careful not to interfere with
    - each other's data
    - each other's use of time
- **Thread class defines two main methods**
  - start   start a new thread
  - run   run this thread
- **a class that wants multiple threads**
  - extends Thread
  - implements run()
  - calls start() when ready, e.g., in constructor

Multi-threaded server

```
public class multiserver {

static String port = "5194";

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

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

}
```

Thread part...

```
class echo extends Thread {
    Socket sock;

echo(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);
    }
}
```