Networking

- Hardware
- Protocols
- Software

Networking Options

Network type	maximum bandwidth (Mbits/second)	latency (microsecs)
Fast Ethernet	100	200
Gigabit Ethernet	1000	29–62
Myrinet 2000	2000–4000	2.6 – 3.2
Myrinet 10G	10000	2.2
Infiniband	10000-20000	1-2
10 Gigabit Ethernet	10000	??

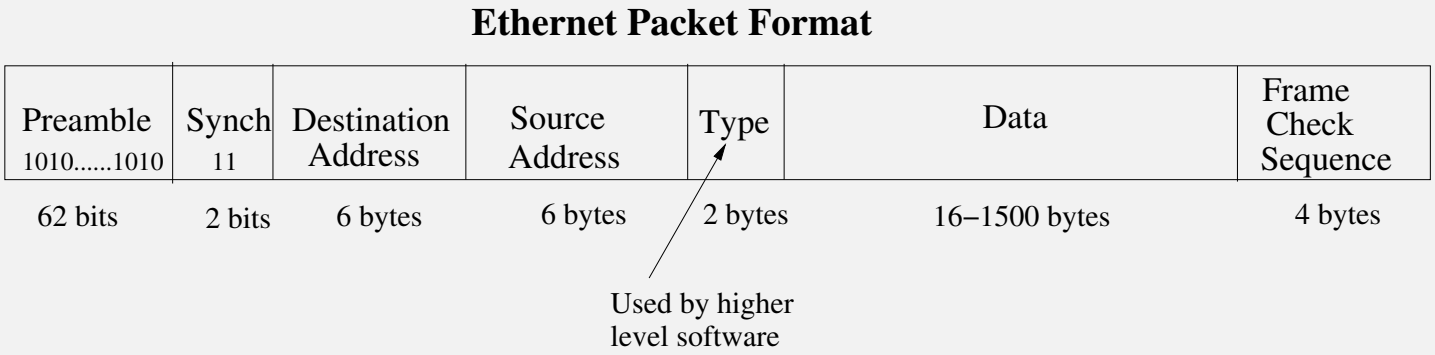
Ethernet is the most cost-effective choice. Ethernet is a packet-based serial multi-drop network requiring no centralized control. All network access and arbitration control is performed by distributed mechanisms.

Ethernet Protocol

- In an Ethernet network, machines use the Carrier Sense Multiple Access with Collision Detect (CSMA/CD) protocol for arbitration when multiple machines try to access the network at the same time.
- If packets collide, then the machines choose a random number from the interval $(0, k)$ and try again.
- On subsequent collisions, the value k is doubled each time, making it a lot less likely that a collision would occur again. This is an example of an *exponential backoff protocol*.

Ethernet Packets

The Ethernet packet has the format shown below.



Note that the Maximum Transmission Unit (MTU) is 1500 bytes. Messages larger than that must be broken into smaller packets by higher layer network software. Higher end switches accept Jumbo packets (up to 9000 bytes), which can improve the performance significantly.
For many network drivers under Linux, the MTU can be set on the fly without reloading the driver!
Use the program ethereal or wireshark to watch Ethernet packets on your network!

Network Topology Design

Ranges of possibilities.

- Shared multi-drop passive cable, or
- Tree structure of hubs and switches, or
- Custom complicated switching technology, or
- One big switch.

Network Topology Options

Hubs and Switches.

- Direct wire. Two machines can be connected directly by a Ethernet cable (usually a Cat 5e cable) without needing a hub or a switch. With multiple NICs per machine, we can create networks but then we need to specify routing tables to allow packets to get through. The machines will end up doing double-duty as routers.
- Hubs and Repeaters All machines are visible from all machines and the CSMA/CD protocol is still used. A hub/repeater receives signals, cleans and amplifies, redistributes to all nodes.
- Switches. Accepts packets, interprets destination address fields and send packets down only the segment that has the destination node. Allows half the machines to communicate directly with the other half (subject to bandwidth constraints of the switch hardware). Multiple switches can be connected in a tree or sometimes other schemes. The root switch can become a bottleneck. The root switch can be a higher bandwidth switch.

Switches

Switches can be **managed** or **unmanaged**. Managed switches are more expensive but they also allow many useful configurations. Here are some examples.

- Port trunking (a.k.a Cisco EtherChannel). Allows up to 4 ports to be treated as one logical port. For example, this would allow a 4 Gbits/sec connection between two Gigabit switches.
- Linux Channel Bonding. Channel bonding means to bond together multiple NICs into one logical network connection. This requires the network switch to support some form of port trunking. Supported in the Linux kernel.
- Switch Meshing. Allows up to 24 ports between switches to be treated as a single logical port, creating a very high bandwidth connection. useful for creating custom complicated topologies.
- Stackable, High bandwidth Switches. Stackable switches with special high bandwidth interconnect in-between the switches. For example, Cisco has 24-port Gigabit stackable switches with a 32 Gbits/sec interconnect. Up to 8 such switches can be stacked together. All the stacked switches can be controlled by one switch and managed as a single switch. If the controlling switch fails, the remaining switches hold an election and a new controlling switch is elected. Baystack also has stackable switches with a 40 Gbits/sec interconnect.

Network Interface Cards

- The Ethernet card, also known as the Network Interface Controller (NIC), contains the Data Link Layer and the Physical Layer (the two lowest layers of networking). Each Ethernet card has a unique hardware address that is know as its MAC address (MAC stands for Media Access Controller). The MAC address is usually printed on the Ethernet card. The command `ifconfig` can be used to determine the MAC address from software.
- NICs with PXE boot support are useful for automated software installation.
- Another issue to consider is that having multi-processor boards may cause more load on the network cards in each node. Certain network cards have dual-network processors in them, making them better candidates for multi-processor motherboards.

Networking Models

- UUCP (Unix to Unix CoPy). Mostly over telephone lines to support mail and USENET news network. UUCP does not support remote login, rpc or distributed file systems.
- The ARPANET Reference Model (ARM) was the network model that led to the ISO OSI seven layer standardized model.
- ISO Open System Interconnection (OSI). A reference model for networking prescribes seven layers of network protocols and strict methods of communication between them. Most systems implement simplified version of the OSI model. The ARPANET Reference Model (ARM) can be seen as a simplified OSI model.

Network Models (contd.)

ISO	ARM	4.2 BSD Layers	Example
application presentation	process applications	user programs/libraries	telnet
session		sockets	sock_stream
transport	host–host	protocols	TCP/IP
network	network interface	network interface	Ethernet driver
data link			
hardware	network hardware	network hardware	interlan controller

TCP/IP Addresses

- The most prevalent protocol in networks is the Internet Protocol (IP). There are two higher-level protocols that run on top of the IP protocol. These are TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).
- IPv4 protocol has 32-bit addresses while the IPv6 protocol has 128-bit addresses. IP address range is divided into networks along an address bit boundary.
- The portion of the address that remains fixed within a network is called the network address and the remainder is the *host address*. The address with all 0's in the host address, for example 192.168.1.0, is the network address and cannot be assigned to any machine. The address with all 1's in the host address, for example 192.168.1.255, is the network broadcast address.
- Three IP ranges are reserved for private networks.
  - 10.0.0.0 – 10.255.255.255
  - 172.16.0.0 – 172.31.255.255
  - 192.168.0.0 – 192.168.255.255

  These addresses are permanently unassigned, not forwarded by Internet backbone routers and thus do not conflict with publicly addressable IP addresses.

Sockets

- A *socket* is an endpoint of communication.
- A socket in use usually has an *address* bound to it.
- The nature of the address depends upon the *communication domain* of the socket.
- Processes communicating in the same domain use the same *address format*.

Typical communication domains:

domain type	symbolic name	address format
Unix domain	`AF_Unix`	pathnames
Internet domain	`AF_INET`	Internet address and port number

Types of Sockets

- *Stream sockets.* Reliable, duplex, sequenced data streams. e.g. pipes, TCP protocol
- *Sequenced packet sockets.* Reliable, duplex, record boundaries
- *Datagram sockets.* Unreliable, unsequenced, variable size packets
- *Reliably delivered message sockets.*
- *Raw sockets*. Allows access to TCP, IP or Ethernet protocol

Socket System Calls

- A socket() system call creates a socket. It returns a *socket descriptor*, similar to a file descriptor. The socket descriptor indexes the open file table.
- For another process to access a socket, it must have a name. A name is bound to a socket by the bind() system call.
- connect(...) Connects a process to a socket with a well-known address (e.g. in the AF_INET domain, the address is the IP address and a port number).
- listen(...) Informs the operating system that the process is willing to accept connections to a socket that has been bound. Has an argument that lets the operating system know how many connections to queue up.
- accept(...) Normally a blocking call. When a connection is made, returns with a new socket id.
- close(...), shutdown(...): Close both or one end of a socket respectively.

Client-Server Setup Using Sockets

Server side	Client side
socket(...)	
bind(...)	socket(...)
listen(...)	connect(...)
accept(...)	
read/write	read/write
close/shutdown	

TCP/IP and Linux/Unix Networking

- Port numbers in the range 1-255 are reserved in TCP/IP protocols for well known servers. In addition, Linux/Unix reserve the ports 1-1023 for superuser processes. Ports from 1024 to 65535 are available for user processes.
- The file /etc/services contains the port numbers for well known servers. For example:
  - port 37 is reserved for getting the time from a system
  - port 7 is used for echoing the data sent by a client back
  - port 21 is used by the FTP (File Transfer Protocol) client/server
  - port 22 is used by SSH (Secure Shell Protocol) client/server
  - port 80 is used by the HTTP (HyperText Transfer Protocol) daemon
  - port 443 is used by the HTTP over TLS/SSL (secure connections) (which is the web server).
- The configuration directory /etc/xinetd.d/ contains several files, one per service type, that control the what is provided by the internet super-daemon xinetd under Linux.

Servers

Design types:

- A single-threaded server handles one connection at a time.
- A multi-threaded server accepts connections and passes them off to their own threads for processing.
- A multi-process server forks off a copy of itself after a connection to handle the client, while the original server process goes back to accept further connections.

Client/Server Examples in C

- Single-threaded server `timeserver.c` and client `timeclient.c`
- Multi-process server: `tcpserver.c` and client `tcpclient.c`
- Multi-threaded server: Left as an exercise!
- Note that read/write on sockets is slightly different than read/write on files. A read/write on a socket may return a count less than asked for. This is not an error since with sockets, the buffer in the kernel may be full. We can just keep calling read/write until the right amount of data has been read or written.

Single-threaded Time Server Example (C/TCP)

```
/* appropriate header files */
int main(int argc, char **argv) {
    int                 listenfd, connfd;
    char                buff[MAXLINE];
    time_t              ticks;
    struct addrinfo hints, *res;

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;     // fill in my IP for me
    getaddrinfo(NULL, "5005", &hints, &res);
    listenfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    if (listenfd < 0) err_quit("Cannot create socket:");
    if (bind(listenfd, res->ai_addr, res->ai_addrlen) < 0)
            err_quit("Bind error:");
    if  (listen(listenfd, LISTENQ) < 0)
            err_quit("Listen error:");
    for (;;) {
        connfd = accept(listenfd, (struct sockaddr *) NULL, NULL);
        if (connfd < 0) err_ret("Accept error: ");
        ticks = time(NULL);
        snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
        write(connfd, buff, strlen(buff));
        close(connfd);
    }
}
```

Time Client Example (C/TCP)

```
/* appropriate header files */
int main(int argc, char **argv) {
    int                 sockfd, n;
    char                recvline[MAXLINE + 1];
    struct addrinfo hints, *res;

    if (argc != 2)
        err_quit("Usage: timeclient <hostname>");
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    getaddrinfo(argv[1], "5005" , &hints, &res);
    if ((sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol)) < 0)
        err_sys("socket error");
    if (connect(sockfd, res->ai_addr, res->ai_addrlen) < 0)
        err_sys("connect error");
    while ((n = read(sockfd, recvline, MAXLINE)) > 0) {
        recvline[n] = 0;    /* null terminate */
        if (fputs(recvline, stdout) == EOF)
            err_sys("fputs error");
    }
    if (n < 0)
        err_sys("read error");
    exit(0);
}
```

Multi-process Server Example (C/TCP)

```
/* appropriate header files */
const char MESSAGE[] = "Hello World\n";
const int BACK_LOG = 5;
int main(int argc, char **argv)
{
    int serverSocket = 0, on = 0, status = 0, childPid = 0;
    char *port;
    struct addrinfo hints, *res;
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <port>\n", argv[0]);
        exit(1);
    }
    port = argv[1];
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;
    getaddrinfo(NULL, "5005", &hints, &res);
    serverSocket = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    if (serverSocket < 0)
        err_quit("Cannot create socket:");
    on = 1;
    status = setsockopt(serverSocket, SOL_SOCKET, SO_REUSEADDR, (char *)&on, sizeof(on));
    if (status == -1) {
        perror("setsockopt(...,SO_REUSEADDR, ...)"); exit(1);
    }
    /* When connection is closed, linger a bit to ensure all data has arrived.  */
    { struct linger linger = { 0 };
        linger.l_onoff = 1;
        linger.l_linger = 30;
        status = setsockopt(serverSocket, SOL_SOCKET, SO_LINGER, (char *) &linger, sizeof(linger));
        if (status == -1) {
            perror("setsockopt(...,SO_LINGER, ...)");
            exit(1);
        }
    }
...
```

Multi-process Server Example (C/TCP) (contd.)

```
...
    if (bind(serverSocket, res->ai_addr, res->ai_addrlen) < 0) {
            err_quit("Bind error:");
    } else {
        fprintf(stderr, "%s: server bound to port %s\n", argv[0], port);
    }
    if  (listen(serverSocket, BACK_LOG) < 0)
            err_quit("Listen error:");
    while (1) {
        struct sockaddr_in clientName = { 0 };
        int slaveSocket, clientLength = sizeof(clientName);
        memset(&clientName, 0,sizeof(clientName));
        slaveSocket = accept(serverSocket, (struct sockaddr *) &clientName, &clientLength);
        if (slaveSocket == -1) err_quit("accept():");

        childPid = fork();
        switch(childPid) {
            case -1:  /* fork failed */
                err_quit("fork()");
            case 0:   /* in child */
                close(serverSocket);
                if (-1 == getpeername(slaveSocket, (struct sockaddr *) &clientName, &clientLength))
                    err_quit("getpeername()");
                else {
                    printf("Connection request from %s\n", inet_ntoa(clientName.sin_addr));
                }
            /* Server application specific code goes here.  */
                write(slaveSocket, MESSAGE, strlen(MESSAGE));
                close(slaveSocket);
                exit(0);
            default:  /* in parent */
                close(slaveSocket);
        }
    }
    exit(0);
}
```

Client (C/TCP)

```
/* appropriate header files */
int main(int argc, char **argv)
{
    int clientSocket, status = 0;
    int len, i;
    struct addrinfo hints, *res;
    char buffer[256] = "";
    char *remoteHost = NULL;
    char *remotePort = NULL;
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <Serverhost> <serverPort>\n", argv[0]);
        exit(1);
    }
    remoteHost = argv[1];
    remotePort = argv[2];

    memset(&hints, 0, sizeof(hints));
    /* AF_INET is for IPv4, AF_INET6 for IPv6, AF_UNSPEC for either */
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    status = getaddrinfo(remoteHost, remotePort, &hints, &res);
    if (status < 0) {
        fprintf(stderr, "%s: %s\n", argv[0], gai_strerror(status));
        exit(1);
    }
    clientSocket = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    if (clientSocket == -1) {
        herror("socket()"); exit(1);
    }
```

Client (C/TCP) contd.

```
...
    status = connect(clientSocket, res->ai_addr, res->ai_addrlen);
    if (status == -1)
        err_quit("Error: connect");

      /* Get message from server */
    /*
       Note that we loop over read since with TCP, we may get a 50 byte
       message from the server as a single message or 5 10 byte messages
       or 50 1 byte messages depending upon the buffering the network
       protocol stack in the kernel.
    */
    while ((status = read(clientSocket, buffer, sizeof(buffer)-1)) > 0) {
        if (status < 0)
            err_quit("read()");
        len = strlen(buffer);
        for (i=0; i<len; i++)
            putchar(buffer[i]);
    }
    exit(0);
}
```

Useful Tools

- Use `netstat -ni` to find information on the network interfaces.
- Use `netstat -rn` to see the routing table.
- Use `netstat -nap` to see the processes that are using specific interfaces and ports. You need to be superuser to be able to see complete process information. Nice way of determining who has a port bound up!
- Use `netstat -s` to see a summary of network statistics. For example, `netstat -s -udp` summarizes all UDP traffic.
- Use `lsof -w -n -i tcp:<port num>` gives the pid of the process using that port so you can kill a process that didn't clean up or properly release the port.
- Use `/sbin/ifconfig eth0` to get details on the interface eth0. Running `ifconfig` without any options gives details on all interfaces.
- Use `ping` to check if a machine is alive. Use `ping -b` with a network address to find all machines on a local area network.
- Use `ethereal` or `wireshark` to watch network packets in real time! You will need superuser access to be able to use ethereal or `wireshark` fully. Great debugging tool.

Client/Server Communication Using Sockets in Java

- A server creates a ServerSocket object for a specific port and uses the accept() method to wait for a connection.
- The client uses (hostname, port number) pair to locate the server.
- The server accepts the client's request and creates a Socket object for communicating with the client. There is a separate Socket object created for each client request accepted.
- Now the server and the client can read/write to the streams associated with the sockets.
- Always open *OutputStream* before *InputStream* on a socket to avoid deadlock and synchronization problems.

Client Example (Java)

```
try {
    Socket server = new Socket("foo.bar.com",1234);
    InputStream in = server.getInputStream();
    OutputStream out = server.getOutputStream();

    out.write(42); // write a byte

    //write a newline or carriage return delimited string
    PrintWriter pout = new PrintWriter(out, true);
    pout.println("Hello!");

    //read a byte
    Byte response = in.read();

    // read a newline or carriage return delimited string
    BufferedReader bin = new BufferedReader (new InputStreamReader(in));
    String answer = bin.readLine();

    //send a serialized Java object
    ObjectOutputStream oout = new ObjectOutputStream(out);
    oout.writeObject(new java.util.Date());
    oout.flush();

    server.close();
} catch (IOException e) {}
```

Server Example (Java)

```
try { // meanwhile, on foo.bar.com...
    ServerSocket listener = new ServerSocket(1234);
    while (!finished) {
        Socket client = listener.accept(); //wait for connection
        InputStream in = client.getInputStream();
        OutputStream out = client.getOutputStream();

        Byte someByte = in.read(); // read a byte

        // read a newline or carriage return delimited string
        BufferedReader bin = new BufferedReader (new InputStreamReader(in));
        String someString = bin.readLine();

        out.write(42); // write a byte
        PrintWriter pout = new PrintWriter(out, true);
        pout.println("Goodbye!");

        //read a serialized Java object
        ObjectInputStream oin = new ObjectInputStream(in);
        Date date = (Date) oin.readObject();

        client.close();
        //...
    }
    listener.close();
} catch (IOException e) {}
```

TCP examples in Java

- A remote date client. DateAtHost.java in `sockets/Java/`.
- Single-threaded server and client: See TimeServer.java and TimeClient.java in sockets/Java/tcp/single-threaded/.
- Multi-threaded server and client: See TimeServer.java and TimeClient.java in sockets/Java/tcp/multi-threaded/.

Remote Date Example (Java)

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

public class DateAtHost extends java.util.Date {
    static int timePort = 37;
    static final long offset = 2208988800L;   // Seconds from century to
                                              // Jan 1, 1970 00:00 GMT

    public DateAtHost(String host, int port) throws IOException {
        Socket server = new Socket(host, port);
        DataInputStream din = new DataInputStream(server.getInputStream());
        int time = din.readInt(); // uses network byte order (big Endian)
        server.close();

        setTime((((1L << 32) + time) - offset) * 1000);
    }
    public DateAtHost(String host) throws IOException {
        this(host, timePort);
    }

    // Example usage:  java DateAtHost emerald.boisestate.edu
    public static void main (String [] args) throws Exception {
        System.out.println(new DateAtHost(args[0]));
    }
}
```

Single-threaded time server

```
import java.io.*;
import java.net.*;
/**
   A single-threaded time server in Java.
   @author amit
*/
public class TimeServer
{
    private InputStream in;
    private OutputStream out;
    private int port = 5005;
    private ServerSocket s;

    public static void main (String args[]) {
        TimeServer server = new TimeServer();
        server.serviceClients();
    }

    public TimeServer() {
        try {
            s = new ServerSocket(port);
            serviceClients();
        } catch (IOException e) {
            System.err.println(e);
        }
    }
    ...
```

```
    ...
    public void serviceClients()
     {
         Socket sock;

         while (true)
         {
             try {
                 sock = s.accept();
                 in = sock.getInputStream();
                 out = sock.getOutputStream();
                 System.out.println("Received connect from " +
                         sock.getInetAddress().getHostAddress());
                 ObjectOutputStream oout = new ObjectOutputStream(out);
                 oout.writeObject(new java.util.Date());
                 oout.flush();
                 Thread.sleep(4000); //4 secs
                 sock.close();
             } catch (InterruptedException e) {
                 System.err.println(e);
             } catch (IOException e) {
                 System.err.println(e);
             }
         }
     }
}
```

Time Client

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

public class TimeClient {
    public static void main (String args[]) {
        if (args.length != 2) {
            System.err.println("Usage: java TimeClient <serverhost> <port>");
            System.exit(1);
        }
        String host = args[0];
        int port = Integer.parseInt(args[1]);
        try {
            Socket s = new Socket(host, port);
            InputStream in = s.getInputStream();
            OutputStream out = s.getOutputStream();
            ObjectInputStream oin = new ObjectInputStream(in);
            Date date = (Date) oin.readObject();
            System.out.println("Time on host "+host+" is "+date);
        } catch (IOException e1) {
            System.out.println(e1);
        } catch (ClassNotFoundException e2) {
            System.out.println(e2);
        }
    }
}
```

Multi-threaded Time Server

```
import java.io.*;
import java.net.*;
/** A multi-threaded time server.  */
public class TimeServer {
    private int port;
    ServerSocket ss;
    public TimeServer(int port) {
        try { ss = new ServerSocket(port);
        } catch (IOException e) { System.err.println(e); }
    }
    public void runServer() {
        Socket client;
        try {
            while (true) {
                client = ss.accept();
                System.out.println("Received connect from " +
                             client.getInetAddress().getHostName() + " [ " +
                             client.getInetAddress().getHostAddress() + " ] ");
                new ServerConnection(client).start();
            }
        } catch  (IOException e) {
            System.err.println(e);
        }
    }
    public static void main (String args[]) {
        if (args.length < 1) {
            System.err.println("Usage: java TimeServer <port>");
            System.exit(1);
        }
        TimeServer server = new TimeServer(Integer.parseInt(args[0]));
        server.runServer();
    }
}
```

```
class ServerConnection extends Thread {
    Socket client;
    ServerConnection (Socket client) throws SocketException {
        this.client = client;
        setPriority(NORM_PRIORITY - 1);
        System.out.println("Created thread "+this.getName());
    }

    public void run() {
        try {
            InputStream in = client.getInputStream();
            OutputStream out = client.getOutputStream();
            ObjectOutputStream oout = new ObjectOutputStream(out);
            oout.writeObject(new java.util.Date());
            oout.flush();
            client.close();
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println(e);
        } catch (IOException e) {
            System.out.println("I/O error " + e);
        }
    }
}
```

Sockets and Security in Java

- The SecurityManager can impose arbitrary restrictions on on applets and applications as to what hosts they may or may not talk to, and whether they can listen for connections.
- The web browser allows socket connections only to the host that served them. Untrusted applets are not allowed to open server sockets themselves.
- A server could run a proxy that lets the applet communicate indirectly with anyone it likes.

Object Based Server/Clients

- The client will send a serialized object to the server. This object represents a request. The server will send an object back as an reply that represents the response.
- We will use a base class `Request` for the various kinds of requests.
  public class Request implements java.io.Serializable
  public class DateRequest extends Request

```
public class WorkRequest extends Request {
    public Object execute() {return null;}
}
```

- The client sends a `WorkRequest` object to the server to get the server to perform work for the client. The server calls the request object's `execute` method and returns the resulting object as a response.

Client Sending WorkRequest Objects

```
public class MyCalculation extends WorkRequest {
    int n;
    public MyCalculation(int n) {
        this.n = n;
    }
    public Object execute() {
        return new Integer(n * n);
    }
}
```

Client Sending WorkRequest Objects (continued)

```
import java.net.*;
import java.io.*;
public class Client {
 public static void main(String argv[]) {
  try {
      Socket server = new Socket(argv[0],Integer.parseInt(argv[1]));
      ObjectOutputStream out =
          new ObjectOutputStream(server.getOutputStream());
      ObjectInputStream in =
          new ObjectInputStream(server.getInputStream());

      out.writeObject(new DateRequest());
      out.flush();
      System.out.println(in.readObject());

      out.writeObject(new MyCalculation(2));
      out.flush();
      System.out.println(in.readObject());
      server.close();
  } catch (IOException e) {
      System.out.println("I/O error " + e); // I/O error
  } catch (ClassNotFoundException e2) {
      System.out.println(e2); // Unknown type of response object
  }
 }
}
```

Object Server

```
import java.net.*;
import java.io.*;
public class Server {
  public static void main(String argv[]) throws IOException {
    ServerSocket ss = new ServerSocket(Integer.parseInt(argv[0]));
    while (true)
        new ServerConnection(ss.accept()).start();
  }
}
```

Object Server (continued)

```
class ServerConnection extends Thread {
  Socket client;
  ServerConnection (Socket client) throws SocketException {
    this.client = client;
    setPriority(NORM_PRIORITY - 1);
  }
  public void run() {
    try {
        ObjectOutputStream out =
            new ObjectOutputStream(client.getOutputStream());
        ObjectInputStream in =
            new ObjectInputStream(client.getInputStream());
        while (true) {
            out.writeObject(processRequest(in.readObject()));
            out.flush();
        }
    } catch (EOFException e3) { // Normal EOF
        try {
            client.close();
        } catch (IOException e) { }
    } catch (IOException e) {
        System.out.println("I/O error " + e); // I/O error
    } catch (ClassNotFoundException e2) {
        System.out.println(e2); // Unknown type of request object
    }
  }
```

Object Server (continued)

```
  private Object processRequest(Object request) {
    if (request instanceof DateRequest)
        return new java.util.Date();
    else if (request instanceof WorkRequest)
        return ((WorkRequest)request).execute();
    else
        return null;
  }
}
```

Running the Object Server/Client

- Start the server on one host:
  on plainoldearth.net: java Server 1234
- Start the client anywhere on the Internet. E.g. on `restaurant.endofuniverse.net`:
  `java Client plainoldearth.net 1234`
- Note that the server machine must have all the classes that the client has in order to be able to execute them on the client's behalf. That may be an unreasonable assumption since you may want to serve many kinds of clients without having to store all their classes.

Socket and ServerSocket Options

- ServerSocket and Socket classes have several useful options.
- For example: we can set a timeout on a socket, we can set the receive buffer sizes etc
- See examples in the folder: `sockate/Java/socket-options/`

Datagram Sockets

- A datagram is a discrete chunk of data transmitted in one chunk.
- Datagrams are not guaranteed to be delivered, nor are they guaranteed to arrive in the right order. Even duplicate datagrams might arrive.
- Datagrams use the UDP protocol, which is more lightweight than the TCP protocol.
- Domain Name Service (DNS) and Network File System (NFS) use UDP.

Datagram Sockets Example

- See example: UdpServer1.java and UdpClient1.java in Java/udp/ex1.

UDP Server

```
public class UdpServer1
{
    private int count;
    private DatagramSocket s;
    private DatagramPacket packet;

    public UdpServer1(int port) {
        try {
            s = new DatagramSocket(port);
            packet = new DatagramPacket(new byte [1024], 1024);
        } catch (Exception e) {
            System.out.println(e);
        }
    }

    public int getCount() { return count; }

    public void runServer() {
        count = 0;
        try {
            while (true) {
                s.receive(packet);
                s.send(packet); // echo back the datagram
                count++;
            }
        } catch (Exception e) {
            System.out.println(e);
        }
    }
...
```

UDP Server (contd).

```
    public static void main(String [] args)
    {
        int port = 0;
        if (args.length != 1) {
            System.err.println("Usage: java UdpServer1 <port#>");
            System.exit(1);
        }
        port = Integer.parseInt(args[0]);
        UdpServer1 server = new UdpServer1(port);
        StatsThread stats = new StatsThread(server);
        Runtime current = Runtime.getRuntime();
        current.addShutdownHook(stats);
        server.runServer();
    }
}

class StatsThread extends Thread
{
    UdpServer1 server;

    public StatsThread(UdpServer1 server)
    {
        this.server = server;
    }

    public void run()
    {
        int count = server.getCount();
        System.err.println("Number of datagrams received by server: "+count);
    }
}
```

UDP Client

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

public class UdpClient1 {

    public static void main(String[] args)
    {
        byte[] data = new byte[1024];
        if (args.length != 2) {
            System.err.println("Usage: java UdpClient1 <serverhost> <port>");
            System.exit(1);
        }
        String myHost = args[0];
        int myPort = Integer.parseInt(args[1]);


        try {
            InetAddress addr = InetAddress.getByName(myHost);
            DatagramPacket packet =
                new DatagramPacket(data, data.length, addr, myPort);
            DatagramSocket ds = new DatagramSocket();
            datagramTest(ds, packet, 2000);
            ds.close();
        } catch (IOException e) {
            System.out.println(e);  // Error creating socket
        }
    }
...
```

UDP Client

```
    public static void datagramTest(DatagramSocket ds,
                                    DatagramPacket packet,
                                    int count)
    {
        byte[] temp = new byte[1024];
        try {
            InetAddress serverAddr = packet.getAddress();
            for (int i=0; i<count; i++) {
                packet.setData(temp);
                ds.send(packet);
                ds.receive(packet);
                if (packet.getAddress().equals(serverAddr)) {
                    System.out.println("recv'd datagram #"+i+
                                       " back from server "+
                                       packet.getAddress().getHostName());
                } else {
                    System.out.println("recv'd datagram back"+
                                       " not from server but from  "+
                                       packet.getAddress().getHostName()+
                                       " (ignored)");
                }
            }
        } catch (IOException e) {
            System.out.println(e);  // Error creating socket
        }
    }
}
```

Example Network Applications

- How to implement a chat program?
- How to implement a telephone program?
- How to implement a videophone program?
- How to implement a web server?

The HTTP Protocol and Web Servers

- A Web Server implements at least the HTTP protocol. In order to talk to a Web server, a client program (e.g. a web browser) must speak the HTTP protocol.
- Details of the HTTP protocol can be found at the home page for the World Wide Web consortium (www.w3.org).
  Requests/Methods in the HTTP Protocol:

```
GET <pathname> HTTP/x.y  (e.g. GET /sample.html HTTP/1.0
HEAD <pathname> (same as GET except only metadata is returned)
POST <string>  (the server should accept the entity enclosed i
               (useful for running CGI-scripts)
```

  Response from server:

```
HTTP-Version status-code reason-phrase <CR><LF>
```

Status codes in the HTTP Protocol

Status codes:

```
200 OK
201 Created

301 Moved permanently
305 Use Proxy
307 Temporary redirect

400 Bad request (bad syntax)
401 Unauthorized
402 Payment required
403 Forbidden
404 Not found

500 Internal server error
501 Not implemented
503 Service unavailable
505 HTTP version not supported
```

A Tiny Web Server

This web server will serve files without any protection from a system.

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

public class TinyHttpd {
    public static void main(String argv[]) throws IOException {
        ServerSocket ss = new ServerSocket(Integer.parseInt(argv[0]));
        System.out.println("starting...");
        while (true) {
            new TinyHttpdConnection(ss.accept()).start();
            System.out.println("new connection");
        }
    }
}

class TinyHttpdConnection extends Thread {
    Socket client;
    TinyHttpdConnection (Socket client) throws SocketException {
        this.client = client;
        setPriority(NORM_PRIORITY - 1);
    }
}
```

A Tiny Web Server (contd.)

```
public void run() {
    try {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(client.getInputStream(), "8859_1"));
        OutputStream out = client.getOutputStream();
        PrintWriter pout = new PrintWriter(
            new OutputStreamWriter(out, "8859_1"), true);
        String request = in.readLine();
        System.out.println("Request: "+request);

        StringTokenizer st = new StringTokenizer(request);
        if ((st.countTokens() >= 2) && st.nextToken().equals("GET")) {
            if ((request = st.nextToken()).startsWith("/"))
                request = request.substring(1);
            if (request.endsWith("/") || request.equals(""))
                request = request + "index.html";
            try {
                FileInputStream fis = new FileInputStream (request);
                byte [] data = new byte [ fis.available() ];
                fis.read(data);
                out.write(data);
                out.flush();
            } catch (FileNotFoundException e) {
                pout.println("404 Object Not Found"); }
        } else { pout.println("400 Bad Request");}
        client.close();
    } catch (IOException e) {System.out.println("I/O error " + e);}
}
```

Using the Built-In Security Manager

Java has a built-in security manager, which if activated gives the same level of access as given to applets (that is, not much). The security manager can be activated with a command line option.
java -Djava.security.manager TinyHttpd
However, we want to give access to create and use sockets. So we create a policy file (using the tool policytool that comes with the Java toolkit).

```
grant {
        permission java.net.SocketPermission
                "*:1024-", "listen,accept,connect";
};
```

Add the following after the catch for FileNotFoundException.

```
catch (SecurityException e) { pout.println("403 Forbidden");}
```

Now, recompile and run the server as follows.
java -Djava.security.manager -Djava.security.policy=mysecurity.policy TinyHttpd 1234

Adding a Custom Security Manager to TinyHttpd

```
 import java.io.*;

class TinyHttpdSecurityManager extends SecurityManager {
    public void checkAccess(Thread g) { };
    public void checkListen(int port) { };
    public void checkLink(String lib) { };
    public void checkPropertyAccess(String key) { };
    public void checkAccept(String host, int port) { };
    public void checkWrite(FileDescriptor fd) { };
    public void checkRead(FileDescriptor fd ) {  };

    public void checkRead(String s) {
        if (new File(s).isAbsolute() || (s.indexOf("..") != -1))
          throw new SecurityException("Access to file: "+s+" denied.");
    }
}
// add the following to the TinyHttpd at the start of the main method
// but after creating the ServerSocket

System.setSecurityManager(new TinyHttpdSecurityManager());
```

Suggestions for Improvement

- Use a buffer and send large amount of data in several passes.
- Generate linked listings for directories (if no index.html was found).
- Log all requests in a log file. A sample entry is shown below (taken from the access log of Apache web server):

```
66.249.68.50 - - [22/Jan/2012:15:16:00 -0700]
"GET /~amit/teaching/455/lab/examples/security/
MD5sums/MD5Test.java HTTP/1.1" 304 - "-" "Mozilla/5.0
(compatible; Googlebot/2.1;
 +http://www.google.com/bot.html)"
```

- Allow applets to communicate via proxies.
- Add other kinds of requests (other than GET)
- Use scalable I/O with `java.nio` package.

Scalable I/O with java.nio package

- *Nonblocking* and *selectable* network communications are used to create services that can handle very high volumes of simultaneous client requests.
- Starting one thread per client request can consume a lot of resources. One strategy is to use nonblocking I/O operations to manage a lot of communications from a single thread. The second strategy is to use a configurable pool of threads, taking advantage of machines with many processors.
- The `java.nio` package provides selectable channels. A *selectable channel* allows for the registration of a special kind of listener called a *selector* that can check the readiness of the channel for operations such as reading and writing or accepting or creating network connections.

Selectable Channels

- Create a selector object. Selector selector = Selector.open();
- To register one or more channels with the selector, set them to nonblocking mode.

```
SelectableChannel channelA = ...;
channelA.configureBlocking(false);
```

- Then, we register the channels.

```
int interestOps = SelectionKey.OP_READ | SelectionKey.OP_WRITE;
SelectionKey key = channelA.register(selector, interestOps);
```

- The possible values of interest ops are: OP_READ, OP_WRITE, OP_CONNECT and OP_ACCEPT. These values can be OR'd together to express interest in one or more operations.
- Once one or more channels are registered with the Selector, we can perform a select operations by using one of the select() methods.

```
int readyCount = selector.select(); //block until one channel is ready
int readyCount = selector.selectNow(); // returns immediately
int readyCount = selector.select(50); // timeout of 50 milliseconds

while (selector.select(50) == 0);
```

Checking for ready channels

Once select() comes back with a non-zero ready count, then we can get the set of ready channels from the Selector with the selectedKeys() method and iterate through them.

```
Set readySet = selector.selectedKeys();
for (Iterator itr = readySet.iterator(); itr.hasNext();) {
    SelectionKey key = (SelectionKey) itr.next();
    itr.remove(); // remove the key from the ready set
    // use the key in the application
}
```

LargerHttpd

- The LargerHttpd is a nonblocking web server that uses SocketChannels and a pool of threads to service requests.
- A single thread executes the main loop that accepts new connections and checks the readiness of existing client connections for reading or writing.
- Whenever a client needs attention, it places the job in a queue where a thread from our thread pool waits to service it.

```
/* appropriate import statemenst */

public class LargerHttpd {
    Selector clientSelector;
    ClientQueue readyClients = new ClientQueue();

    public void run(int port, int threads) throws IOException {
        clientSelector = Selector.open();
        ServerSocketChannel ssc = ServerSocketChannel.open();
        ssc.configureBlocking(false);
        InetSocketAddress sa =
            new InetSocketAddress(InetAddress.getLocalHost(), port);
        ssc.socket().bind(sa);
        ssc.register(clientSelector, SelectionKey.OP_ACCEPT);

        for (int i=0; i<threads; i++)
            new Thread() { public void run() {
                while (true) try { handleClient(); } catch (IOException e) { }
            } }.start();

        while (true) try {
            while (clientSelector.select(50) == 0);
            Set readySet = clientSelector.selectedKeys();
            for(Iterator it = readySet.iterator(); it.hasNext();) {
                SelectionKey key = (SelectionKey)it.next();
                it.remove();
                if (key.isAcceptable())
                    acceptClient(ssc);
                else {
                    key.interestOps(0);
                    readyClients.add(key);
                }
            }
        } catch (IOException e) { System.out.println(e); }
    }
...
```

```
...
    void acceptClient(ServerSocketChannel ssc) throws IOException {
        SocketChannel clientSocket = ssc.accept();
        clientSocket.configureBlocking(false);
        SelectionKey key =
            clientSocket.register(clientSelector, SelectionKey.OP_READ);
        HttpdConnection client = new HttpdConnection(clientSocket);
        key.attach(client);
    }

    void handleClient() throws IOException {
        SelectionKey key = (SelectionKey)readyClients.next();
        HttpdConnection client = (HttpdConnection)key.attachment();
        if (key.isReadable())
            client.read(key);
        else
            client.write(key);
    }

    public static void main(String argv[]) throws IOException {
        new LargerHttpd().run(Integer.parseInt(argv[0]), 3);
    }
}
```

```
class HttpdConnection {
    static Charset charset = Charset.forName("8859_1");
    static Pattern httpGetPattern = Pattern.compile("(?s)GET /?(\\S*).*");
    SocketChannel clientSocket;
    ByteBuffer buff = ByteBuffer.allocateDirect(64*1024);
    String request;
    String response;
    FileChannel file;
    int filePosition;

    HttpdConnection (SocketChannel clientSocket) {
        this.clientSocket = clientSocket;
    }

    void read(SelectionKey key) throws IOException {
        if (request == null && (clientSocket.read(buff) == -1
                || buff.get(buff.position()-1) == '\n'))
            processRequest(key);
        else
            key.interestOps(SelectionKey.OP_READ);
    }

  ...
```

```
...
    void processRequest(SelectionKey key) {
        buff.flip();
        request = charset.decode(buff).toString();
        Matcher get = httpGetPattern.matcher(request);
        if (get.matches()) {
            request = get.group(1);
            if (request.endsWith("/") || request.equals(""))
                request = request + "index.html";
            //System.out.println("Request: "+request);
            try {
                file = new FileInputStream (request).getChannel();
            } catch (FileNotFoundException e) {
                response = "404 Object Not Found";
            }
        } else
            response = "400 Bad Request" ;

        if (response != null) {
            buff.clear();
            charset.newEncoder().encode(
                CharBuffer.wrap(response), buff, true);
            buff.flip();
        }
        key.interestOps(SelectionKey.OP_WRITE);
    }
...
```

```
...
    void write(SelectionKey key) throws IOException {
        if (response != null) {
            clientSocket.write(buff);
            if (buff.remaining() == 0)
                response = null;
        } else if (file != null) {
            int remaining = (int)file.size()-filePosition;
            long got = file.transferTo(filePosition, remaining, clientSocket);
            if (got == -1 || remaining <= 0) {
                file.close();
                file = null;
            } else
                filePosition += got;
        }
        if (response == null && file == null) {
            clientSocket.close();
            key.cancel();
        } else
            key.interestOps(SelectionKey.OP_WRITE);
    }
}

class ClientQueue extends ArrayList {
    synchronized void add(SelectionKey key) {
        super.add(key);
        notify();
    }
    synchronized SelectionKey next() {
        while (isEmpty())
            try { wait(); } catch (InterruptedException e) { }
        return (SelectionKey)remove(0);
    }
}
```