Machine vs. transport endpoints

- IP packets address only the machine
  - IP header identifies source IP address, destination IP address
  - IP address is a 32-bit address that refers to a machine

- IP packet delivery is not guaranteed to be reliable or in-order

- Transport-level protocols on top of IP:
  - TCP/IP
  - UDP/IP
  - Port numbers

---

TCP/IP

- Virtual circuit service
- Sends ACK for each received packet
- Checksum to validate data
- Data may be transmitted simultaneously in both directions
- No record markers but data arrives in sequence

UDP/IP

- Datagram service
- Packet may be lost
- Data may arrive out of sequence
- Checksum for data but no retransmit

---

Sockets

Create a generalized IPC model

- Dominant API for transport layer connectivity
- Operating system interface to the network
- Created at UC Berkeley for 4.2BSD Unix (1983)
- Design goals
  - Communication between processes should not depend on whether they are on the same machine
  - Communication should be efficient
  - Interface should be compatible with files
  - Support different protocols and naming conventions
    - Sockets is not just for the Internet Protocol family

---

What is a socket?

Abstract object from which messages are sent and received

- Looks like a file descriptor

- Application can select particular style of communication
  - Virtual circuit (connection-oriented), datagram (connectionless), message-based, in-order delivery

- Unrelated processes should be able to locate communication endpoints
- Sockets can have a name
- Name should be meaningful in the communications domain
  - E.g., Address & port for IP communications
How are sockets used?

Client: web browser
Server: web server

![Diagram of socket operations]

Connection-Oriented (TCP) socket operations

Client
Create a socket
Name the socket (assign local address, port)
Wait for and accept a connection; get a socket for the connection
Read/write byte streams
Close the socket

Server
Create a socket
Name the socket (assign local address, port)
Listen for connections
Read/write byte streams
Close the listening socket

Connectionless (UDP) socket operations

Client
Create a socket
Name the socket (assign local address, port)
Send a message
Receive a message
Close the socket

Server
Create a socket
Name the socket (assign local address, port)
Receive a message
Send a message
Close the socket

Using sockets in Java

- java.net package
  - Socket class
    - Deals with sockets used for TCP/IP communication
  - ServerSocket class
    - Deals with sockets used for accepting connections
  - DatagramSocket class
    - Deals with datagram packets (UDP/IP)

- Both Socket and ServerSocket rely on the SocketImpl class to actually implement sockets
  - But you don’t have to think about that as a programmer

Create a socket for listening: server

Server:
- create, name, and listen are combined into one method
  - ServerSocket constructor

```
ServerSocket svc = new ServerSocket(80, 5);
```

Several other flavors (see api reference)

1. Server: create a socket for listening

Client: web browser
Server: web server

```
ServerSocket svc = new ServerSocket(80, 5);
```

Send HTTP request message to get a page
Receive HTTP request message
Process HTTP request
Send HTTP response message
Receive HTTP response message
Display a page
Server: wait for (accept) a connection

- accept method of ServerSocket
  - block until connection arrives
  - return a Socket

```java
ServerSocket svc = new ServerSocket(80, 5);
Socket req = svc.accept();
```

Create a socket: client

Client:
- create, name, and connect operations are combined into one method
- Socket constructor

```java
Socket s = new Socket("www.rutgers.edu", 2211);
```

Several other flavors (see api reference)

2. Server: wait for a connection (blocking)

Client: web browser
Server: web server

```java
ServerSocket svc = new ServerSocket(80);
Socket req = svc.accept();
```

Exchange data

- Obtain InputStream and OutputStream from Socket
  - layer whatever you need on top of them
    - e.g. DataInputStream, PrintStream, BufferedReader, ...

Example:
```java
client
DataInputStream in = new DataInputStream(s.getInputStream());
PrintStream out = new PrintStream(s.getOutputStream());

server
DataInputStream in = new BufferedReader(new InputStreamReader(req.getInputStream()));
String line = in.readLine();
DataOutputStream out = new DataOutputStream(req.getOutputStream());
out.writeBytes(mystring + "\n");
```
TCP vs. UDP sockets
• TCP ("stream sockets")
  – Requires a connection (connection-oriented)
  – Dedicated socket for accepting connections
  – Communication socket provides a bi-directional link
  – Byte-stream: no message boundaries
• UDP ("datagram sockets")
  – Connectionless: you can just send a message
  – Data send in discrete packets (messages)
The sockets system call interface (What the operating system gives us)

<table>
<thead>
<tr>
<th>System call</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket</td>
<td>Create a socket</td>
</tr>
<tr>
<td>bind</td>
<td>Associate an address with a socket</td>
</tr>
<tr>
<td>listen</td>
<td>Set the socket to listen for connections</td>
</tr>
<tr>
<td>accept</td>
<td>Wait for incoming connections</td>
</tr>
<tr>
<td>connect</td>
<td>Connect to a socket on the server</td>
</tr>
<tr>
<td>read/write,</td>
<td>Exchange data</td>
</tr>
<tr>
<td>sendto/recvfrom,</td>
<td></td>
</tr>
<tr>
<td>sendmsg/recvmsg</td>
<td></td>
</tr>
<tr>
<td>close/shutdown</td>
<td>Close the connection</td>
</tr>
</tbody>
</table>

**POSIX system call interface**

**Step 1 (client & server)**
Create a socket

```
int s = socket(domain, type, protocol)
```

- **AF_INET** is for IPv4
- **AF_INET6** is IPv6
- **AF_BTH** is Bluetooth

**SOCK_STREAM**:
- reliable, in-order, 2-way.
- TCP/IP

**SOCK_DGRAM**:
- datagrams (UDP/IP)

**SOCK_RAW**:
- "raw" – allows app to modify the network layer header

Conceptually similar to open BUT
- open creates a new reference to a possibly existing object
- `socket` creates a new instance of an object

**Step 2 (client & server)**
Name the socket (assign address, port)

```
int error = bind(s, addr, addrlen)
```

Naming for an IP socket is the process of assigning our address to the socket.
- The address is the full transport address: the IP address of the network interface as well as the UDP or TCP port number

**Step 3a (server)**
Set socket to be able to accept connections

```
int error = listen(s, backlog)
```

**Step 3 (client)**
Connect to server

```
int error = connect(s, svraddr, svraddrlen)
```

The client can send a connection request to the server once the server did a listen and is waiting for accept.
Step 3b (server)
Wait for a connection from client

```c
int snew = accept(s, clntaddr, &clntalen);
```

This tells you where the socket came from: full transport address.

New socket for this communication session:
Block the process until an incoming connection comes in.

Step 4. Exchange data

**read/write** system calls (same as for file systems)

**send/recv** system calls

```c
int send(int s, void *msg, int len, uint flags);
int recv(int s, void *buf, int len, uint flags);
```

Block the process until an incoming connection comes in.

Step 5
Close connection

```c
shutdown(s, how);
```

how:

- SHUT_RD (0): send but not receive
- SHUT_WR (1): send more data
- SHUT_RDWR (2): send and receive (=0+1)

You can use the regular close system call too, which does a complete shutdown, the same as shutdown(s, SHUT_RDWR).

Java provides shortcuts that combine calls

**Example**

```java
Socket s = new Socket("www.rutgers.edu", 2211);
sock = socket(AF_INET, SOCK_STREAM, 0);
myaddr.sin_family = AF_INET;
myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
myaddr.sin_port = htons(0);
bind(sock, (struct sockaddr *)&myaddr, sizeof(myaddr));

struct hostent *hp = gethostbyname("www.rutgers.edu");
if (connect(fd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
    /* connect failed */
}
```

Sample Client-Server Program

- To illustrate programming with TCP/IP sockets, we'll write a small client-server program:
  - Client:
    - Read a line of text from the user
    - Send it to the server; wait for a response (single line)
    - Print the response
  - Server:
    - Wait for a connection from a client
    - Read a line of text
    - Return a response that contains the length of the string and the string converted to uppercase
    - Exit
Sample Client-Server Program

- We will then embellish this program to:
  - Have a continuously-running server
  - Allow a client to send multiple lines of text
  - Make the server multi-threaded so it can handle concurrent requests
  - Specify a host on the command line

Classes for input/output

- With Java, you’ll often layer different input/output stream classes depending on what you want to do.
- Here are some common ones:

  **Input**
  - InputStream
  - BufferedReader
  - InputStreamReader

  **Output**
  - OutputStream
  - DataOutputStream
  - PrintStream
  - DataOutputStream

Handling output

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputStream</td>
<td>The basics – write a byte or a bunch of bytes</td>
</tr>
<tr>
<td>DataOutputStream</td>
<td>Allows you to write Unicode (multibyte) characters, booleans, doubles, floats, ints, etc. Watch out if using this because the other side might not be Java and might represent the data differently. The two most useful things here are writeBytes(String s), which writes a string out as a bunch of 1-byte values and writeBytes(b, int off, int len), which writes a sequence of bytes from a byte array.</td>
</tr>
<tr>
<td>PrintStream</td>
<td>Allows you to use print and println to send characters. Useful for line-oriented output.</td>
</tr>
<tr>
<td>FilterOutputStream</td>
<td>Needed for PrintStream. On its own, just gives you the same write capabilities you get with OutputStream</td>
</tr>
</tbody>
</table>

Handling input

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputStream</td>
<td>The basics – read a byte or a bunch of bytes</td>
</tr>
<tr>
<td>BufferedReader</td>
<td>Buffers input and parses lines. Allows you to read data a line at a time via readLine(). You can also use read(char[] buf, int off, int len) to read characters into a portion of an array.</td>
</tr>
<tr>
<td>InputStreamReader</td>
<td>You need this to use BufferedReader. It converts bytes (that you’ll be sending over the network) to Java characters.</td>
</tr>
</tbody>
</table>

Client: step 1

- Read a line of text from the standard input (usually keyboard)
  - We use readLine to read the text. For that, we need to use the BufferedReader class on top of the InputStreamReader on top of the system input stream (System.in)

```java
import java.io.*;

public class line {
  public static void main(String args[]) throws Exception {
    String line;
    BufferedReader userdata = new BufferedReader(new InputStreamReader(System.in));
    line = userdata.readLine();
    System.out.println("got: "+ line);
  }
}
```

Test #1

- Don’t hesitate to write tiny programs if you’re not 100% sure how something works

```java
public class line {
  public static void main(String args[]) throws Exception {
    String line;
    BufferedReader userdata = new BufferedReader(new InputStreamReader(System.in));
    line = userdata.readLine();
    System.out.println("got: "+ line);
  }
}
```

- Notice that readLine) removes the terminating newline character from a line
  - If we want to send line-oriented text, we’ll need to suffix a newline ("\n") to the string
### Client: step 2

- Establish a socket to the server, send the line, and get the result
  - Create a socket.
  - For now, we will connect to ourselves – the name "localhost" resolves to our local address.
  - For now, we will hard-code a port number: 12345

- Get input and output streams from the socket
  - The methods `getInputStream()` and `getOutputStream()` return the basic streams for the socket
  - Create a `DataOutputStream` for the socket so we can write a string as bytes
  - Create a `BufferedReader` so we can read a line of results from the server

```java
Socket sock = new Socket("localhost", 12345);    // create a socket and connect
DataOutputStream toServer = new DataOutputStream(sock.getOutputStream());
BufferedReader fromServer = new BufferedReader(new InputStreamReader(sock.getInputStream()));
String line = userInput.readLine(); // read a line from the user
// send the line to the server
toServer.writeBytes(line + '\n');
String result = fromServer.readLine(); // read a one-line result
System.out.println(result); // print it
sock.close(); // and we're done
```

---

### Client: step 3

- Send the line we read from the user and read the results

```java
String result = fromServer.readLine(); // read the response from the server
System.out.println(result);
```

---

### Our client – version 1

But we can’t test it yet because we don’t have the server!

```java
import java.io.*;
import java.net.*;
public class TCPClient {
  public static void main(String[] args) throws Exception {
    String line;
    BufferedReader userdata = new BufferedReader(new InputStreamReader(System.in));
    Socket sock = new Socket("localhost", 12345);
    DataOutputStream toServer = new DataOutputStream(sock.getOutputStream());
    BufferedReader fromServer = new BufferedReader(new InputStreamReader(sock.getInputStream()));
    line = userdata.readLine(); // read a line from the user
    toServer.writeBytes(line + '\n'); // send the line to the server
    String result = fromServer.readLine(); // read a one-line result
    System.out.println(result); // print it
    sock.close(); // and we're done
  }
}
```

---

### Server: step 1

- Create a socket for listening
  - This socket’s purpose is only to accept connections
  - Java calls this a `ServerSocket`
  - For now, we’ll use a hard-coded port: 12345
  - If the port number is 0, the operating system will assign a port.
  - The backlog is the maximum queue length for unserviced arriving connections
  - The backlog is missing or 0, a default backlog will be used

```java
import java.net.*;
public class wait {
  public static void main(String[] args) throws Exception {
    ServerSocket svc = new ServerSocket(12345, 5);    // listen on port 12345
    Socket conn = svc.accept();   // get a connection
  }
}
```

---

### Server: step 2

- Wait for a connection
  - This method will block until a connection comes in
  - When a client connects to port 12345 on this machine, the `accept()` method will return a new socket that is dedicated to communicating to that specific client

```java
Socket conn = svc.accept();  // get a connection
```

---

### Test #2

- We can now test that a client can connect to the server
- Let’s write a tiny server that just waits for a connection and then exits

```java
import java.net.*;
public class wait {
  public static void main(String[] args) throws Exception {
    ServerSocket svc = new ServerSocket(12345, 5);    // listen on port 12345
    Socket conn = svc.accept();  // get a connection
  }
}
```

- Now run the client in another window
  - As soon as the client starts, it will establish a connection and the server will exit
Server: step 3

- Get input/output streams for the socket
  - We will create a `BufferedReader` for the input stream so we can use `readLine` to read data a line at a time
  - We will create a `DataOutputStream` for the output stream so we can write bytes.

```java
// get the input/output streams for the socket
BufferedReader fromClient = new BufferedReader(new InputStreamReader(conn.getInputStream()));
DataOutputStream toClient = new DataOutputStream(conn.getOutputStream());
```

Server: step 4

- Read a line of data from the client (via `fromClient`)
- Create the result
- Write the result to the client (via `writeBytes`)

```java
String line = fromClient.readLine();
System.out.println("got line " + line);
String result = line.length() + " : " + line.toUpperCase() + '
';
toClient.writeBytes(result);
```

Server: step 5

- Done! Close the socket
  - Close the socket to the client to stop all communication with that client
  - Close the listening socket to disallow any more incoming connections. Servers often run forever and therefore we often will not do this.

```java
System.out.println("server exiting
");
conn.close();
svc.close();
```

Test #3

- Compile TCPServer.java and TCPClient.java
  - javac * .java
- In one window, run
  - java TCPServer
- In another window, run
  - java TCPClient
- The client will wait for input. Type something
  - Hello
- It will respond with the server’s output:
  - 5: HELLO

Version 2

- We don’t want the server to exit
- Instead, have it wait for another connection
- Simple:
  - Create the ServerSocket
  - Then put everything else in a forever loop (for(;;))
  - Never close the ServerSocket
- Now we can keep the server running and try running the client multiple times
### Our server – version 2

```java
public class TCPServer {
    public static void main(String[] args) throws Exception {
        ServerSocket svc = new ServerSocket(12345, 5);  // listen on port 12345
        for (;;) {
            Socket conn = svc.accept();     // get a connection from a client
            BufferedReader fromClient = new BufferedReader(new InputStreamReader(conn.getInputStream()));
            DataOutputStream toClient = new DataOutputStream(conn.getOutputStream());
            String line = fromClient.readLine();    // read the data from the client
            System.out.println("got line \" + line + \"\n");       // do the work
            String result = line.length() + " : " + line.toUpperCase() + '\n';       // do the work
            toClient.writeBytes(result);    // send the result
            System.out.println("closing the connection\n");
            conn.close();           // close connection
        }
    }
}
```

### Version 3: let’s support multiple lines

- Instead of having the server close the connection when a single line of text is received, allow the client to read multiple lines of text
  - Each line is sent to the server; the response is read & printed
  - An end of file from the user signals the end of user input
- This is typically control-D on Mac/Linux/Unix systems (see the `stty` command)

### Version 3 – server changes

- We need to change the server too
  - Read lines from a socket until there are no more
  - When the client closes a socket and the server tries to read, it will get an end-of-file (``null``) will return a null
- A simple loop lets us iterate over the lines coming in from one client

```java
while ((line = fromClient.readLine()) != null) {
    // while there's data from the client
    // do work on the data
}
```

### The server handles only one connection

- Run the server in one window
- Run the client in another window
  - Type a bunch of text
  - Each line produces a response from the server
- Run the client again in yet another window
  - Type a bunch of text
  - Nothing happens. There’s no connection to the server!
  - You have to exit the first client before this one can connect.
- We need to make the server multi-threaded

### Version 4 – add multi-threading to the server

- We define the server to implement Runnable
- Define a constructor: called for each new thread

```java
public class TCPServer implements Runnable {
    Socket conn;  // if this is a per-thread copy of the client socket
    TCPServer(Socket sock) {
        this.conn = sock;  // store the socket for the connection
    }
    @Override
    public void run() {
        // server logic
    }
}
```
Version 4 – add multi-threading to the server

• The main function just gets connections and creates threads

```java
public static void main(String[] args) throws Exception {
    ServerSocket svc = new ServerSocket(12345, 5);  // listen on port 12345
    for (;;) {
        Socket conn = svc.accept();     // get a connection from a client
        System.out.println("got a new connection");
        new Thread(new TCPServer(conn)).start();
    }
}
```

This creates the thread's state and calls the constructor.
This creates the thread of execution and calls run() in the thread.
When run returns, the thread exits.

The per-connection work is done in the thread

```java
public void run() {
    try {
        BufferedReader fromClient = new BufferedReader(new InputStreamReader(conn.getInputStream()));
        DataOutputStream toClient = new DataOutputStream(conn.getOutputStream());
        String line;
        while ((line = fromClient.readLine()) != null) {        // while there's data from the client
            System.out.println("got line "+ line);
            String result = line.length() + " : " + line.toUpperCase() + '
';       // do the work
            toClient.writeBytes(result);    // send the result
        }
        System.out.println("closing the connection
");
        conn.close();           // close connection and exit the thread
    } catch (IOException e) {
        System.out.println(e);
    }
}
```

Version 5

• Allow the client to specify the server name on the command line
  – If it’s missing, use "localhost"

```java
public class TCPClient {
    public static void main(String[] args) throws Exception {
        String server = "localhost"; // default server
        BufferedReader userdata = new BufferedReader(new InputStreamReader(System.in));
        if (args.length > 1) {
            System.err.println("usage: java TCPClient server_name");
            System.exit(1);
        } else if (args.length == 1) {
            server = args[0];
            System.out.println("server = " + server);
        }
        Socket sock = new Socket(server, 12345);     // connect to localhost port 12345
```

The end