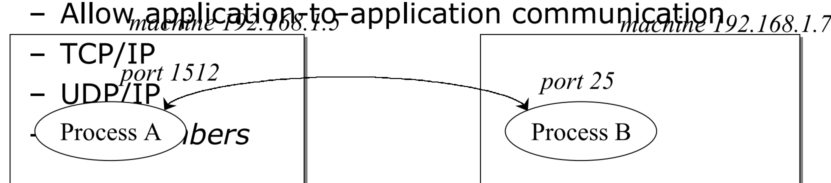
Sockets and the operating system

CS 417

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:
  - Allow application-to-application communication
  - TCP/IP
  - UDP/IP



CS 417

(This part is a review from the lecture)

The IP protocol allows packets to be sent from one machine to another machine. Each IP packet header identifies the source and destination machines by their 32-bit IP address.

The assumption at the IP layer is that the delivery of packets is unreliable (packets may get lost or data may get corrupted) and may not be in-order (packets may arrive in a different order than they were sent.

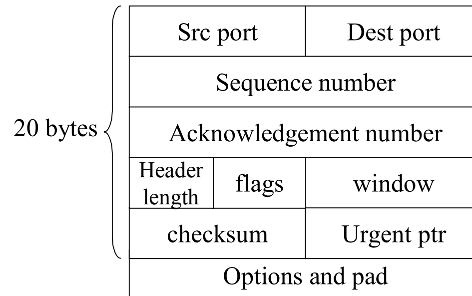
Two transport-level protocols allow us to deal with application-to-application communication. These are TCP/IP and UDP/IP.

Both TCP and UDP use **port numbers** to identify transport endpoints (applications). A port number is a 16-bit number that is associated with an application.

For example, if Process A (which grabbed use of port number 1512) communicates with Process B (which grabbed the use of port 25), an IP packet from process A to process B will be addressed with a source address of 192.168.1.5 and a destination address of 192.168.1.7. The TCP or UDP header, which is encapsulated *within* the IP data will contain a source port of 1512 and a destination port of 25.

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



TCP header

CS 417

TCP/IP – Transport Control Protocol

Provides virtual circuit service – connection oriented – you set up a connection and just read and write data on that connection. No need to address each packet.

Packets are acknowledged by the protocol – no chance of packet loss

A checksum is present to validate data – no data corruption. If there is, an acknowledgement is not sent and the packet is retransmitted

TCP does not maintain record boundaries. For example, if you perform three writes: *5 bytes, 20 bytes, 15 bytes*, the other side *may* receive all the data in a single *read* of *40 bytes*. Or… it may not. If you want to preserve these boundaries, it is up to you to layer your own protocol on top of this. For example:

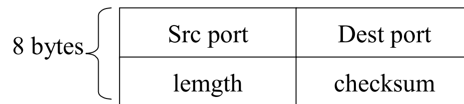
send *(length, data), (length, data), …*

Instead of

data, data, data, …

UDP/IP

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



Src port	Dest port
lemgth	checksum

UDP header

UDP/IP – User Datagram Protocol

Provides datagram service – connectionless– each packet must be addressed to the destination

Data may be lost: it is up to the application to discover that a packet did not arrive at its destination. Application can send acknowledgements if this is needed.

Data may arrive out of sequence: if packets take a different route via IP they may arrive in a different order from the one sent. TCP buffers up out-of-order packets in the kernel and presents them in the right order to the user.

A checksum for the data exists in the UDP header. If data corruption is detected then the packet is dropped. To the application, this is the same as a lost packet.

UDP is a lighter weight protocol: header is 8 bytes vs. 20+ bytes for TCP. Linux source for udp is around 1200 lines vs. >8000 lines for tcp.

More importantly, dealing with UDP traffic uses less kernel resources: state about the connection, receive buffers to deal with out-of-order data, retransmission does not have to be handled.

Sockets and ports

- Sockets are a popular communication abstraction
  - Allow us to address data to a destination address and port number
  - Allow us to bind our socket to a port number and source address.
- *bind*
  - Assign source address and port number
- *connect*
  - Assign destination address and port number

CS 417

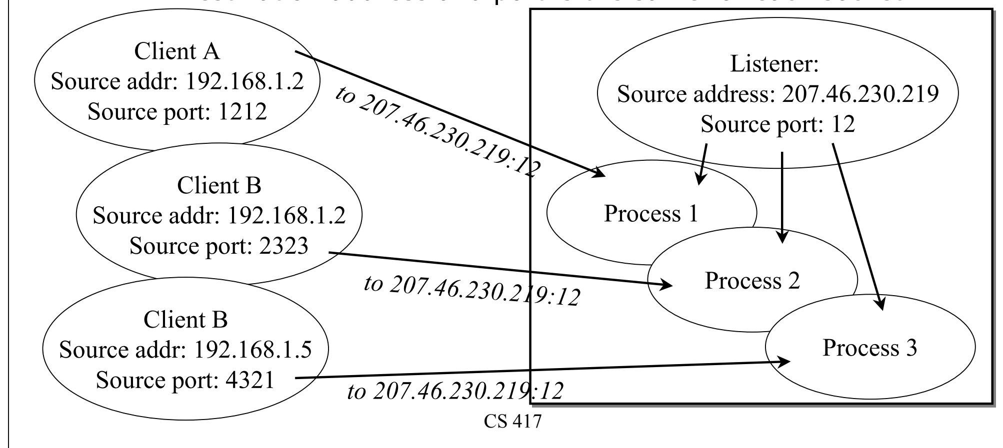
As we've seen with sockets programs, sockets are a popular implementation for achieving network communication.

After creating a socket, we use the *bind* system call to assign it to an address and port number. Typically, this address is 0.0.0.0, which represents INADDR_ANY, or any valid incoming address (particularly useful if a machine has several connections and several IP addresses). For servers, the port number is usually chosen by the user so that other programs will be able to specify it. For clients, one typically asks the operating system to pick any unused port number (this is requested by setting the port to 0 in *bund*).

For connection-oriented protocols (TCP), the server address and port number is specified with the *connect* system call. For connectionless protocols (UDP), the address and port number may be specified explicitly when sending data using *sendto* or *sendmsg* system calls.

Multiple senders may send from the same address, port!

- Server
  - Sets socket for *listen* and waits in *accept*
  - *accept* returns a new socket
  - Typically, a new process or thread will handle that session
  - Destination address and port is the same for each socket!



When a server set up a socket for listening and waits for an incoming connection with the *accept* call, a new socket is returned for each accepted connection.

A server will typically create a new thread or fork a new process that will be responsible for handling that particular communication session.

Each client still addresses data to the same IP address and port number. Now that data has to be routed to the correct socket so that the right process or thread can get the data and all the data streams do not become interleaved.

Sorting it out: routing data to the correct socket

- Protocol Control Block
  - Table
  - Each entry contains:
    - Local address
    - Local port
    - Foreign address
    - Foreign port
    - Is the socket used for listening?
    - Reference to the socket (file descriptor)

How does the operating system get incoming data to the correct socket when multiple sockets may have the same incoming address and port number?

The operating system maintains a structure called the **Protocol Control Block** (PCB).

The PCB is a table of entries, where each PCB entry contains the following information:

- local address: IP address bound to the socket
- local port: port number bound to the socket
- foreign address: IP address from the remote side
- foreign port: port number from the remote side
- is the socket used for listening?
- reference to the socket descriptor (file descriptor)

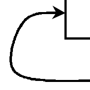
Each socket descriptor contains a pointer to an entry in the PCB table.

socket()

- *socket* system call
  - Allocate a new entry in PCB table

Client:

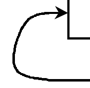
local address	local port	foreign address	foreign port	listen?
0	0	0	0	



new entry in PCB table

Server:

local address	local port	foreign address	foreign port	listen?
0	0	0	0	



new entry in PCB table

The first thing we do in using sockets is to allocate a new socket with the *socket* system call. This causes a new, empty, entry to be created in the PCB table.

bind()

- *bind* system call
  - Assign local address, port

Client:

	local address	local port	foreign address	foreign port	listen?
s →	0.0.0.0	7801	0	0	

Server:

	local address	local port	foreign address	foreign port	listen?
s →	0.0.0.0	1234	0	0	

The bind system call assigns a source address to a socket. This causes the local address and local port fields in the corresponding PCB entry (for that socket) to be filled in.

Suppose the client binds to address 0.0.0.0 (INADDR_ANY), port 7801

Suppose the server binds to address 0.0.0.0 (INADDR_ANY), port 1234

listen()

- *listen* system call
  - Set socket for receiving connections

Client:

	local address	local port	foreign address	foreign port	listen?
s →	0.0.0.0	7801	0	0	

Server:

	local address	local port	foreign address	foreign port	listen?
s →	0.0.0.0	1234	0	0	YES

The *listen* system call is used by a server to set a socket for receiving future *connect* requests. This causes a flag to be set in the PCB entry for that socket

connect()

- *connect* system call
  - Send a *connect* request to server

Client: send a *connect* control message [from 135.250.68.43, port 7801]

	local address	local port	foreign address	foreign port	listen?
s →	0.0.0.0	7801	0	0	

Server: waits for control messages in *accept*, creates new PCB entry

	local address	local port	foreign address	foreign port	listen?
s →	0.0.0.0	1234	0	0	YES
snew →	192.11.35.15	1234	135.250.68.43	7801	

CS 417

The *connect* system call is used by a client to establish a connection to the server.

The client gets the remote address and port as part of the address structure passed to *connect* .

It then sends a *connect* control message to the server. The server should be waiting in *accept* to receive this message.

When the server receives a control message,

it scans the PCB table for an entry where the connect request **matches the local address (0 matches anything) and the socket is in the listen state (I.e., set to receive control messages)**.

- a new socket is allocated and a new PCB entry is created that is associated with the new socket.

- this new PCB entry is populated with the local address on which the connection was received and the foreign address, which is the source address of the *connect* request.

connect() – part 2

- *connect* system call
  - Send a *connect* request to server

Client: send a *connect* control message [from 135.250.68.43, port 7801]

	local address	local port	foreign address	foreign port	listen?
s →	0.0.0.0	7801	192.11.35.15	1234	

Server: waits for control messages in *accept*, creates new PCB entry

	local address	local port	foreign address	foreign port	listen?
s →	0.0.0.0	1234	0	0	YES
snew →	192.11.35.15	1234	135.250.68.43	7801	

CS 417

The server sends a message to the client acknowledging acceptance of the connection. This message contains the real address and port of the server that the client puts in its PCB entry for that socket.

Each message from the client is tagged either as a **data** message or a **control** message (e.g. connect).

If the message is a data message:

search through the list of PCB entries for one where the foreign address and foreign port match the incoming message

If the message is a control message:

search through the list of PCB entries for one where *listen* is set (and foreign address = foreign port = 0).

Programming with sockets

CS 417
Distributed Systems Design

Sockets

- Goals
  - inter-process communication using the same mechanism whether the processes are on the same machine or a different machine
  - efficient
  - compatibility with legacy processes that read from and write to files
  - support for different protocols and naming conventions
    - not limited to TCP/IP and UDP/IP
- Socket
  - an object for sending and receiving messages

Sockets are an attempt at creating a generalized IPC (inter-process communication) model.

They were developed by Berkeley in 1982 as part of the Berkeley version of Unix (BSD 4.1a, with general availability in BSD 4.2 in September 1983).

The design goals were:

- have a uniform mechanism at the operating system level for inter-process communication, independent of whether the processes are on the same machine.
- the interface should be as efficient as possible
- any applications that do not know about networking or IPC should be able to be given input and output file descriptors and just use them without changes for network communication.

  For example - another process may accept or initiate the communication, create the “file” descriptors, and then hand off the descriptors to a process (e.g., by fork & exec) that is network ignorant
- the sockets mechanism should work for a just about any naming scheme and networking protocol - it is not meant only for TCP/IP and UDP/IP.

Programming operations

Client		Server	
1.	Create a socket	1.	Create a socket
2.	Name a socket	2.	Name a socket
		3a.	Set the socket for listening
3c.	Connect to a remote endpoint	3b.	Accept a connection
4.	Communicate	4.	Communicate
5.	Close the connection	5.	Close the connection

not needed for connectionless communication

CS 417

The system-call level interface consists of the following operations:

1. Both the client and server need to create a socket.
2. Since the socket identifies the *transport endpoint* for the application, the transport address will need to be assigned - this is known as *naming* the socket. For IP communication, this means assigning a port number to the socket.

3a. A server has to indicate that it wants to accept connections on this socket.

3b. A server will then block (go to sleep), waiting for incoming connections.

3c. A client will connect to a server.

4. Each party will communicate by sending and receiving messages.
5. When done, each party will close the connection.

For connectionless protocols, steps 3a-c are not needed. A client can just send a message to a server and a server can block while waiting to receive a message. It is up to the server now to disambiguate messages from different client.

Unix system call interface

- System calls:
  - socket
  - bind
  - listen
  - accept
  - connect
  - read/write, send/recv, sendto/recvfrom, sendmsg/recvmsg
  - close/shutdown

Create a socket

`socket` system call:

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

parameters:

`domain`:	identifies address family e.g. AF_INET for IP, AF_UNIX for local, AF_NS for Xeroxs Network Systems
`type`:	type of service required by application SOCK_STREAM - virtual circtuit SOCK_DGRAM - datagram SOCK_RAW - raw IP access
`protocol`:	specify specific protocol. Used if address family supports, say, two versions of virtual circuit service

return: a small integer representing the socket (file descriptor)

Name a socket

bind system call:

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

parameters:

s:	socket descriptor returned by *socket()*
addr:	address structure (`struct sockaddr *`)
addrlen:	length of address structure

return: error code

The *bind* system call allows us to assign a transport endpoint to the socket.

Analogy: **socket**: request a phone line

bind: request a phone number for the line

For IP services, this is where we specify the port number on which we want to accept connections. Since the client will not be accepting connections, it generally will not care what port number it gets, so it can specify a port of 0, in which case the operating system will assign it an available port number.

Note - the client needs a port number because messages from the server back to the client have to be addressed as well.

bind is a separate system call because some communication domains may not require it. Also, some communication domains may require other custom operations to take place on the socket before the binding is performed.

Server: set socket for listening

`listen` system call:

```
int error = bind(s, backlog);
```

parameters:

`s`:	socket descriptor returned by *socket()*
`backlog`:	queue length for pending connections

return: error code

CS 417

The *listen* system call is used by the server to indicate that it will accept connection requests on that socket.

Once done, the operating system will accept such connection requests on behalf of this socket.

The server will generally be sleeping, waiting for a connection request (via the *accept* system call - next slide). When it gets the connection, it will usually fork off another process or create/dispatch a thread to handle that request and go back to listening for more connections. Since there is a delay between accepting a connection request and looping back to accept another one, the *listen* system call allows one to specify a **backlog** - a queue size for pending connections.

Server: accept connections

`accept` system call:

```
int snew = accept(s, clntaddr, addrlen);
```

parameters:

`s`:	socket descriptor returned by *socket()*
`clntaddr`:	`struct sockaddr *` contains returned client address information
`addrlen`:	length of address information

return: a new socket to be used for this communication session

CS 417

The *accept* system call blocks (by default) and waits until a connection comes in on the socket.

Accept creates a **new socket** on which this communication session will take place. The original socket, created with the *socket* system call on the server, is used *only* to get connections.

In addition to returning a new socket, *accept* also returns information about the address of the connecting client (could be used for authentication checking…).

Client: connect

`connect` system call:

```
int error = connect(s, svraddr, addrlen);
```

parameters:

`s`:	socket descriptor returned by *socket()*
`svraddr`:	`struct sockaddr *` contains address of server
`addrlen`:	length of address information

return: error code

The *connect* system call is used by the client to connect to the server.

s - socket returned by the socket() system call

svraddr - structure containing the address of the server. The contents depend on the transport being used

addrlen - length of the address structure. This is passed in because C does not support polymorphism and the address structure length is a function of the transport used.

Exchange data

for connection-oriented service

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

send/recv system calls

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

sendto/recvfrom system calls

```
int sendto(int s, void *msg, int len, uint flags,
       struct sockaddr *to, int tolen);
int recvfrom(int s, void *buf, int len, uint flags,
       struct sockaddr *from, int *fromlen)
```

sendmsg/recvmsg system calls

```
int sendmsg(int s, struct msghdr *msg, uint flags);
int recvmsg(int s, struct msghdr *msg, uint flags);
```

CS 417

Data can now be exchanged between client and server.

The client will treat its socket, *s*, as a file descriptor and use ordinary *read/write* system calls on it.

The server will treat its communication socket, *snew*, as a file descriptor and also use ordinary *read/write* system calls on it.

Alternatively, the client and server may choose to use **send/recv** system calls, which support flags that may be useful for certain communication transports, such as:

- process out of band data
- bypass routing - use direct interface
- don't block
- don't generate a SIGPIPE signal when either end breaks the connection

For **connectionless service**, ***accept*** and ***listen*** were not invoked by the server and ***connect*** was not invoked by the client. The server just receives messages from anyone who sends them. The system calls **sendto/recvfrom** and **sendmsg/recvmsg** allow addresses to be specified on the send line, along with other information. [see man pages]

Stop all further communication

file-sytem **`close`** system call:

```
close(s);
```

or

`shutdown` system call:

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

where

s: socket

how: 0: further receives disallowed
1: further sends disallowed
2: further sends and receives disallowed

A socket can be closed with the normal file-system-based **close** system call or, for greater control, the **shutdown** call can be used. This allows one to disallow certain types of operations (sends or receives).

Using sockets in java

- java.net package
  - **Socket** class
    - deals with sockets used for communication
  - **ServerSocket** class
    - deals with sockets used for accepting connections
  - **DatagramSocket** class
    - deals with datagram packets

  - both Socket and ServerSocket rely on the SocketImpl class to actually implement sockets

Creating a socket

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

host port

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

  - several other flavors (see api reference)
- Server:
  - *create*, *name*, and *listen* are combined into one method
  - **ServerSocket** constructor

port backlog

```
ServerSocket svc = new ServerSocket(2211, 5);
```

  - several other flavors (see api reference)

Server: accept a connection

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

```
ServerSocket svc = new ServerSocket(2211, 5);
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:

client

```
DataInputStream in = new DataInputStream(s.getInputStream());
PrintStream out = new PrintStream(s.getOutputStream());
```

server

```
DataInputStream in = new BufferedReader(
             new InputStreamReader(req.getInputStream()));
DataOutputStream out = new DataOutputStream(
                           req.getOutputStream());
```

Close the sockets

- Close input and output streams first:

client:

```
try {
        out.close();
        in.close();
        s.close();
} catch (IOException e) {}
```

server:

```
try {
        out.close();
        in.close();
        req.close();    // close connection socket
        svc.close();    // close ServerSocket
} catch (IOException e) {}
```

Sample programs

http://www.cs.rutgers.edu/~pxk/rutgers/src/socketdemo.tar

run:

    tar xvf SocketDemo.tar

will create two directories:

    jsocketdemo: java sample program
    socketdemo: C sample program

each directory contains a README file.