COS 316
Precept:
Socket Programming
High-level Architecture

- **Application**
  - Read data from and write data to the socket
  - Interpret the data (e.g., render a Web page)

- **Transport**
  - Deliver data to the destination socket
  - Based on the destination port number (e.g., 80)

- **Internet**
  - Deliver data packet to the destination host
  - Based on the destination IP address

- **Network Access**
  - Transmit data between devices
  - Encapsulate IP packet into frames transmitted by the network
  - Map IP addresses into physical addresses
Terminology

- **IP (IPv4) Addresses**
  - Hosts mapped to 32 bit IP addresses: `aaaaaaaa.bbbbbbbb.cccccccc.dddddd`
  - E.g., `128.112.136.51`
  - Various special IP addresses, e.g., `127.0.0.1`

- **Domain names**
  - IP addresses are mapped to an identification string
  - E.g., `www.cs.princeton.edu`
  - E.g., `localhost`

- **Port** - a unique communication end point on a host, named by a 16-bit integer, and associated with a process

- **Connections**
  - A process on one host communicates with another process on another host over a connection
  - Clients and servers communicate by sending streams of bytes over connections
  - E.g., using TCP or UDP

- **Socket** - end-point of a connection
  - Sending message from one process to another
    - Message must traverse the underlying network
  - Process sends and receives through a “socket”
    - In essence, the doorway leading in/out of the house
  - Socket as an Application Programming Interface
    - Supports the creation of network applications

- **Stream Socket (TCP - Transmission Control Protocol)**
  - Stream of bytes
  - Reliable
  - Connection-oriented

- **Datagram Socket (UDP - User Datagram Protocol)**
  - Collection of messages
  - Best effort
  - Connectionless
Socket Identification

• Receiving host
  • Destination **address** that uniquely identifies host
  • **IP address**: 32-bit quantity

• Receiving socket
  • Host may be running many different processes
  • Destination **port** that uniquely identifies socket
  • **Port number**: 16-bits
Client - Server Communication

• Client “sometimes on”
  • Initiates a request to the server when interested
  • E.g., Web browser on your laptop or cell phone
  • Doesn’t communicate directly with other clients
  • Needs to know server’s address

• Server is “always on”
  • Handles services requests from many client hosts
  • E.g., Web server for the www.cnn.com Web site
  • Doesn’t initiate contact with the clients
  • Needs fixed, known address
Knowing What Port Number To Use

• Popular applications have well-known ports
  • E.g., port 80 for Web and port 25 for e-mail
  • See http://www.iana.org/assignments/port-numbers

• Well-known vs. ephemeral ports
  • Server has a well-known port (e.g., port 80)
    • Between 0 and 1023 (requires root to use)
  • Client picks an unused ephemeral (i.e., temporary) port
    • Between 1024 and 65535

• “5 tuple” uniquely identifies traffic between hosts
  • Two IP addresses and two port numbers
  • + underlying transport protocol (e.g., TCP or UDP)
Using Ports to Identify Services

Client host

Service request for 128.2.194.242:80 (i.e., the Web server)

Server host 128.2.194.242

Web server (port 80)

Echo server (port 7)

OS

Client

Service request for 128.2.194.242:7 (i.e., the echo server)

OS

Web server (port 80)

Echo server (port 7)
Worksheet
Stream Sockets (TCP): Connection-oriented

**Server**

- `socket()`: Create a socket
- `bind()`: Bind the socket (what port am I on?)
- `listen()`: Listen for client (Wait for incoming connections)
- `accept()`: Accept connection
- `recv()`: Receive Request
- `send()`: Send response

**Client**

- `socket()`: Create a socket
- `connect()`: Connect to server
- `send()`: Send the request, data (request)
- `recv()`: Receive response
- `recv()`: Receive response
Datagram Sockets (UDP): Connectionless

Server

- `socket()`: Create a socket
- `bind()`: Bind the socket
- `recvfrom()`: Receive Request
- `sendto()`: Send response

Client

- `socket()`: Create a socket
- `bind()`: Bind the socket
- `sendto()`: Send the request
- `recvfrom()`: Receive response

Data flow:
- Data (request) from Client to Server
- Data (reply) from Server to Client
Example C Server and Client
Byte Order

• Network byte order
  • Big Endian

• Host byte order
  • Big Endian or Little Endian

• Functions to deal with this
  • `htons()` & `hton1()` (host to network short and long)
  • `ntohs()` & `ntoh1()` (network to host short and long)

• When to worry?
  • putting data onto the wire
  • pulling data off the wire
Server: Server Preparing its Socket

• Create a socket
  • \texttt{int socket(int domain, int type, int protocol)}

• Bind socket to the local address and port number
  • \texttt{int bind(int sock\_fd, struct sockaddr *server\_address, socklen\_t addrlen)}
Server: Allowing Clients to Wait

• Many client requests may arrive
  • Server cannot handle them all at the same time
  • Server could reject the requests, or let them wait

• Define how many connections can be pending
  • \texttt{int listen(int socket\_fd, int backlog)}
  • Arguments: socket descriptor and acceptable backlog
  • Returns a 0 on success, and -1 on error
  • Listen is \texttt{non-blocking}: returns immediately

• What if too many clients arrive?
  • Some requests don’t get through
  • The Internet makes no promises…
  • And the client can always try again
Server: Accepting Client Connection

• Now all the server can do is wait…
  • Waits for connection request to arrive
  • **Blocking** until the request arrives
  • And then accepting the new request

• Accept a new connection from a client
  • `int accept(int sockfd,struct sockaddr *addr, socketlen_t *addrlen)`
    • Arguments: sockfd, structure that will provide client address and port, and length of the structure
    • Returns descriptor of socket for this new connection
Client and Server: Closing Connection

• Once the connection is open
  • Both sides and read and write
  • Two unidirectional streams of data
  • In practice, client writes first, and server reads
  • … then server writes, and client reads, and so on

• Closing down the connection
  • Either side can close the connection
  • … using the `int close(int sockfd)`

• What about the data still “in flight”
  • Data in flight still reaches the other end
  • So, server can `close()` before client finishes reading
Server: One Request at a Time?

• Serializing requests is inefficient
  • Server can process just one request at a time
  • All other clients must wait until previous one is done
  • What makes this inefficient?

• May need to time share the server machine
  • Alternate between servicing different requests
    • Do a little work on one request, then switch when you are waiting for some other resource (e.g., reading file from disk)
    • “Nonblocking I/O”
  • Or, use a different process/thread for each request
    • Allow OS to share the CPU(s) across processes
  • Or, some hybrid of these two approaches
Handle Multiple Clients using `fork()`

• Steps to handle multiple clients
  • Go to a loop and accept connections using `accept()`
  • After a connection is established, call `fork()` to create a new child process to handle it
  • Go back to listen for another socket in the parent process
  • `close()` when you are done.

• Want to know more?
  • Checkout out *Beej's guide to network programming*
Sockets in Go
The **net** package

- **net.Listen** receives the ip, port, and protocol, and returns a **net.Listener**

- **net.Listener#Accept** waits for connections from clients
  - Once a client connects, **net.Accept** returns a **net.Conn** to be used for communication

- **net.Dial** connects to the given ip and port, with the specified protocol.
  - Once it is connected, **net.Dial** returns a **net.Conn** to be used for communication
Socket Server/Client: Go

SERVER

• `socket, err := net.Listen("tcp4", "127.0.0.1:8080")`
  • `net.Listen` performs the C `socket`, `bind` and `listen` system calls
  • `socket` is of type `net.Listener`

• `connection, err := server.Accept()`
  • `net.Accept` accepts an incoming client request
  • `connection` is of type `net.Conn`

CLIENT

• `connection, err := net.Dial("tcp4", "127.0.0.1:8080")`
  • Creates a TCP socket, establish connection
  • `connection` is of type `net.Conn`
net.Conn

- `net.Conn.Read` reads from the connection
  - Wrap the connection in `bufio.Reader`

- `net.Conn.Write` writes to the connection

- `net.Conn.Close` closes the connection
net/http (Useful in Future)

- A collection of useful functions for handling and processing http requests