Topics for this Week

• Layered Network Architecture
  – ISO/OSI Reference Model

• Internet Protocol Suite Overview

• Application Programming Interface
  – BSD Socket API

• Readings
  – Sections 1.1-1.5, 6.1.3 (socket programming), 7.3 (world wide web)
  – Tutorials on Socket Programming
Fundamental Problems in Networking

• What can go wrong?
  – Bit-level errors: due to electrical interferences
  – Packet-level errors: packet loss due to buffer overflow/congestion
  – Out of order delivery: packets may take different paths
  – Link/node failures: cable is cut or system crash

• What can be done?
  – Add redundancy to detect and correct erroneous packets
  – Acknowledge received packets and retransmit lost packets
  – Assign sequence numbers and reorder packets at the receiver
  – Sense link/node failures and route around failed links/nodes

• Goal: to fill the gap between what applications expect and what underlying technology provides
Layered Architecture

- *Layering* simplifies the architecture of complex system
- Layer N relies on *services* from layer N-1 to provide a *service* to layer N+1
- *Interfaces* define the services offered
- Service required from a lower layer is independent of it’s implementation
  - Layer N change doesn’t affect other layers
  - Information/complexity hiding
  - Similar to object oriented methodology
Protocols

• *Protocol*: rules by which network elements communicate
• Protocols define the agreement between peering entities
  – The *format* and the *meaning* of messages exchanged
• Protocols in everyday life
  – Examples: traffic control, open round-table discussion etc
Protocols and Services

• Protocols are used to implement services
  – Peering entities in layer N provide service by communicating with each other using the service provided by layer N-1

• Logical vs. physical communication
ISO/OSI and Internet Reference Models

Internet Protocol Stack

OSI Protocol Stack
ISO/OSI Reference Model

- **Application layer**
  - Examples: smtp, http, ftp etc
    - Process-to-process communication
  - All layers exist to support this layer

- **Presentation layer**
  - Conversion of data to common format
    - Example: Little endian vs. big endian byte orders
      - [http://www.cs.umass.edu/~verts/cs32/endian.html](http://www.cs.umass.edu/~verts/cs32/endian.html)
ISO/OSI Reference Model (cont’d)

• Session layer (OSI only)
  – Session setup (authentication)
  – Recovery from failure (broken session)

• Transport layer
  • Examples: TCP, UDP, NCP
  – End-to-end delivery
    • End-host to end-host communication
  – Flow/Error control
ISO/OSI Reference Model (cont’d)

• Network layer
  • Examples: IP, IPX
    – Naming and addressing
    – Routing of packets within a network
    – Avoidance of congested/failed links
ISO/OSI Reference Model (cont’d)

• Data link layer
  • Examples: Ethernet, PPP
    – Data transfer between neighboring elements
  • Framing and error/flow control
  • Media access control

• Physical layer
  – Transmitting raw bits (0/1) over wire
Protocol Packets

- Protocol data units (PDUs): packets exchanged between peer entities
- Service data units (SDUs): packets handed to a layer by an upper layer
- Data at one layer is encapsulated in packet at a lower layer
  - Envelope within envelope: PDU = SDU + (optional) header or trailer
Implementation of Layers
Comments on Layering

• Advantages
  – Modularization eases maintenance and updating

• Drawbacks?
  – Which layer should implement what functionality?
    • Hop-by-hop basis or end-to-end basis
  – Duplication of functionality between layers
    • Error recovery at link layer and transport layer
Internet Protocol “Zoo”
- iSCSI Protocol over IPSec
• iSCSI over SSL
The Internet Network layer

Routing protocols
- path selection
- RIP, OSPF, BGP

IP protocol
- addressing conventions
- packet handling conventions

ICMP protocol
- error reporting
- router “signaling”

Transport layer: TCP, UDP

Network layer

Link layer

physical layer
Internet Protocol (IP)

• Universal service in a heterogeneous world
  – IP over everything
• Virtual overlay network
• Globally unique logical address for a host
• Address resolution
  – logical to physical address mapping
Internet Protocol

• Connectionless unreliable datagram service
• Packets carry a source and a destination address
• Each packet routed independently
• No guarantee that network will not lose packets
  • Error recovery is up to end-to-end protocols
Transport between Neighbors

• Using underlying link layer transmission mechanism
  – Example: Ethernet, Token Ring, PPP

• Mapping from logical IP address to physical MAC address
  – Address Resolution Protocol (ARP)
End to End Transport Protocols

TCP service:
- *connection-oriented:* setup required between client, server
- *reliable transport* between sender and receiver
- *flow control:* sender won’t overwhelm receiver
- *congestion control:* throttle sender when network overloaded

UDP service:
- unreliable data transfer between sender and receiver
- does not provide: connection setup, reliability, flow control, congestion control

Q: Why UDP?
Internet Philosophy

• Network provides **barebones** service
  – Connectionless unreliable datagram by IP
• **Value-added functions** performed “end to end”
  – Error recovery and flow control by TCP
• End user/application knows better
  – Packet loss may be tolerable for voice

• Also known as **“end-to-end argument”**
Client:  
- initiates contact with server ("speaks first")  
- typically requests service from server  

Server:  
- provides requested service to client 

Typical network app has two pieces: *client* and *server*
The Web: The HTTP Protocol

hypertext transfer protocol

- Web’s application layer protocol
- client/server model
  - **client**: browser that requests, receives, “displays” Web objects
  - **server**: Web server sends objects in response to requests
Interprocess Communication

• Within a single system
  – Pipes, FIFOs
  – Message Queues
  – Semaphores, Shared Memory

• Across different systems
  – BSD Sockets
  – Transport Layer Interface (TLI)

• Reference
  – *Unix Network Programming* by Richard Stevens
BSD Socket API

• Introduced in 1981 BSD 4.1 UNIX
• Function call interface to network services
  • system and library calls
    – Network application programming primitives

• Connects two sockets on separate hosts
  – Sockets are owned by processes
  – Processes communicate through sockets
BSD Sockets and Internet Protocols

• API: BSD Sockets
  – Socket: source/destination IP addresses + port numbers

• Transport: TCP/UDP
  – TCP: in-order, reliable data transfer
    • Connection-oriented
  – UDP: unreliable data transfer
    • No connection set-up

• Network: IP
  – Connectionless, no guarantees
Sockets: Conceptual View
Connection-Oriented Application

1. Server gets ready to service clients
   – Creates a socket
   – Binds an address to the socket
     • Server’s address should be made known to clients

2. Client contacts the server
   – Creates a socket
   – Connects to the server
     • Client has to supply the address of the server

3. Accepts connection requests from clients

4. Further communication is specific to application
Creating a socket

int  socket(int family, int service, int protocol)

• **family**: symbolic name for protocol family  
  – AF_INET, AF_UNIX
• **type**: symbolic name for type of service  
  – SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
• **protocol**: further info in case of raw sockets  
  – typically set to 0

Returns *socket descriptor*
Binding Socket with an Address

int bind(int sd, struct sockaddr *addr, int len)

- **sd**: socket descriptor returned by socket()
- **addr**: pointer to sockaddr structure containing address to be bound to socket
- **len**: length of address structure

Returns 0 if success, -1 otherwise
Specifying Socket Address

```c
struct sockaddr_in { 
    short sin_family;       /* set to AF_INET */
    u_short sin_port;       /* 16 bit port number */
    struct in_addr sin_addr; /* 32 bit host address */
    char sin_zero[8];       /* not used */
};

struct in_addr { 
    u_long s_addr;          /* 32 bit host address */
};
```
Bind Example

```c
int sd;
struct sockaddr_in ma;
sd = socket(AF_INET, SOCK_STREAM, 0);

ma.sin_family = AF_INET;
ma.sin_port = htons(5100);
ma.sin_addr.s_addr = htonl(INADDR_ANY);
if (bind(sd, (struct sockaddr *) &ma, sizeof(ma)) != -1)
...
```
Connecting to Server

int connect(int sd, struct sockaddr *addr, int len)
• **sd**: socket descriptor returned by socket()
• **addr**: pointer to sockaddr structure containing server’s address (IP address and port)
• **len**: length of address structure

Returns 0 if success, -1 otherwise
int sd;
struct sockaddr_in sa;
sd = socket(AF_INET, SOCK_STREAM, 0);
sa.sin_family = AF_INET;
sa.sin_port = htons(5100);
sa.sin_addr.s_addr = inet_addr("128.101.34.78");
if (connect(sd, (struct sockaddr *) &sa, sizeof(sa)) != -1)
...

Connect Example
Connection Acceptance by Server

int accept(int sd, struct sockaddr *from, int *len)
• sd: socket descriptor returned by socket()
• from: pointer to sockaddr structure which gets filled with client’s address
• len: length of address structure

Blocks until connection requested or error
• returns a new socket descriptor on success
Connection-oriented Server

```c
int    sd, cd, calen;
struct sockaddr_in    ma, ca;

sd = socket(AF_INET, SOCK_STREAM, 0);
ma.sin_family = AF_INET;
ma.sin_port = htons(5100);
ma.sin_addr.s_addr = htonl(INADDR_ANY);
bind(sd, (struct sockaddr *) &ma, sizeof(ma));

listen(sd, 5);
calen = sizeof(ca);

cd = accept(sd, (struct sockaddr *) &ca, &calen);
…read and write to client treating cd as file descriptor…
```
More on Socket Descriptor

• A 5-tuple associated with a socket
  – \{protocol, local IP address, local port, remote IP address, remote port\}
    • socket() fills the protocol component
    • local IP address/port filled by bind()
    • remote IP address/port by accept() in case of server
    • in case of client both local and remote by connect()

• Complete socket is like a file descriptor
  – Both send and recv through same socket

• Accept returns a new complete socket
  – Original one can be used to accept more connections
sockid = socket()

bind()

listen()

newsockid = accept()

create a child process, fork(), to handle communication (provide service) to client

parent process

child process

child communications read() / write() with client and provides service via newsockid

close(newsockid) and exit()
Streams and Datagrams

• Connection-oriented reliable byte stream
  – SOCK_STREAM based on TCP
  – No message boundaries
  – *Multiple writes* may be consumed by *one read*

• Connectionless unreliable datagram
  – SOCK_DGRAM based on UDP
  – Message boundaries are preserved
  – *Each sendto* corresponds to *one recvfrom*
Input/Output Multiplexing

• Polling
  – Nonblocking option using fcntl()/ioctl()
  – Waste of computer resources
• Asynchronous I/O
  – Generates a signal on an input/output event
  – Expensive to catch signals
• Wait for multiple events simultaneously
  – Using select() system call
  – Process sleeps till an event happens
Select System Call

```c
int select(int maxfdp1, fd_set *readfds,
           fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout)
```

- **maxfdp1**: largest numbered file descriptor + 1
- **readfds**: check if ready for reading
- **writefds**: check if ready for writing
- **exceptfds**: check for exceptional conditions
- **timeout**: specifies how long to wait for events
Timeout in Select

• Wait indefinitely till there is an event
  – Pass `NULL` to the `timeout` argument

• Don’t wait beyond a fixed amount of time
  – Pass pointer to a `timeval` structure specifying the number of seconds and microseconds.

• Just poll without blocking
  – Pass pointer to a `timeval` structure specifying the number of seconds and microseconds as `0`
Working with File Descriptor Set

• Set is represented by a **bit mask**
  – Keep a descriptor **in/out** the set, **turn on/off** corresponding bit
    • Using FD_ZERO, FD_SET and FD_CLR
    • Use FD_ISSET to check for membership

• Example:
  – Make descriptors 1 and 4 members of the readset
    ```c
    fd_set   readset;
    FD_ZERO(&readset);
    FD_SET(1, &readset);
    FD_SET(4, &readset);
    ```
  – Check if 4 is a member of readset
    ```c
    FD_ISSET(4, &readset);
    ```
Return Values from Select

- Arguments `readfds` etc are `value-result`
- Pass set of descriptors you are interested in
- Select modifies the descriptor set
  - Keeps the bit on if an event on the descriptor
  - Turns the bit off if no event on the descriptor
- On return, test the descriptor set
  - Using FD_ISSET
Select Example

```c
fd_set   readset;
FD_ZERO(&readset);
FD_SET(0, &readset);
FD_SET(4, &readset);
select(5, &readset, NULL, NULL, NULL);
if  (FD_ISSET(0, &readset) {
    /* something to be read from 0 */
}
if  (FD_ISSET(4, &readset) {
    /* something to be read from 4 */
}
```
Servers and Services

• Mapping between names and addresses (DNS)
  – Host name to address: gethostbyname()
  – Host address to name: gethostbyaddr()
  – Try using command host
    • Example: “host mail.cs.umn.edu” or “host 128.101.35.200”

• Mapping from a service to a port number
  – Use getservbyname()
  – Look at /etc/services or try ypcat services
Operations on Socket

- getpeername()
  - returns remote address part of socket tuple
- getsockname()
  - returns local address part of socket tuple
- getsockopt()
  - extracts the socket settings
- setsockopt()
  - changes the settings that control socket behavior
Utility Functions

- `inet_addr()`, `inet_ntoa()`
  - dotted decimal string to/from 32-bit address
- `htonl()`, `htons()`, `ntohl()`, `ntohs()`
  - byte ordering functions
- `bcopy()`, `bzero()`, `bcmp()`
  - byte operations
  - avoid using string operations such as `strcpy`