UNIX Sockets

COS 461 Precept 1
Socket and Process Communication

The interface that the OS provides to its networking subsystem
Delivering the Data: Division of Labor

• Network
  – Deliver data packet to the destination host
  – Based on the destination IP address

• Operating system
  – Deliver data to the destination socket
  – Based on the destination port number (e.g., 80)

• Application
  – Read data from and write data to the socket
  – Interpret the data (e.g., render a Web page)
Socket: End Point of Communication

• 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
Two Types of Application Processes Communication

• **Datagram Socket (UDP)**
  – Collection of messages
  – Best effort
  – Connectionless

• **Stream Socket (TCP)**
  – Stream of bytes
  – Reliable
  – Connection-oriented
## User Datagram Protocol (UDP): Datagram Socket

<table>
<thead>
<tr>
<th>UDP</th>
<th>Postal Mail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single socket to receive messages</td>
<td>Single mailbox to receive letters</td>
</tr>
<tr>
<td>No guarantee of delivery</td>
<td>Unreliable</td>
</tr>
<tr>
<td>Not necessarily in-order delivery</td>
<td>Not necessarily in-order delivery</td>
</tr>
<tr>
<td>Datagram – independent packets</td>
<td>Letters sent independently</td>
</tr>
<tr>
<td>Must address each packet</td>
<td>Must address each mail</td>
</tr>
</tbody>
</table>

**Example UDP applications**

Multimedia, voice over IP (Skype)
Transmission Control Protocol (TCP): Stream Socket

TCP
- Reliable – guarantee delivery
- Byte stream – in-order delivery
- Connection-oriented – single socket per connection
- Setup connection followed by data transfer

Telephone Call
- Guaranteed delivery
- In-order delivery
- Connection-oriented
- Setup connection followed by conversation

Example TCP applications
Web, Email, Telnet
**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

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

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

- In UNIX, everything is like a file
  - All input is like reading a file
  - All output is like writing a file
  - File is represented by an integer file descriptor

- API implemented as system calls
  - E.g., connect, send, recv, close, ...
Client-Server Communication
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
- `recv()`: Receive response
Client-Server Communication
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:
- Request (data) from Client to Server
- Reply (data) from Server to Client
Client: Learning Server Address/Port

• Server typically known by name and service
  – E.g., "www.cnn.com" and "http"

• Need to translate into IP address and port #
  – E.g., "64.236.16.20" and "80"

• Get address info with given host name and service
  – `int getaddrinfo(` `char *node,`
         `char *service,`
         `struct addrinfo *hints,`
         `struct addrinfo **result)`

    – *node: host name (e.g., "www.cnn.com") or IP address
    – *service: port number or service listed in `/etc/services` (e.g. ftp)
    – hints: points to a `struct addrinfo` with known information
Client: Learning Server Address/Port (cont.)

- Data structure to host address information

```c
struct addrinfo {
    int      ai_flags;
    int      ai_family;  // e.g. AF_INET for IPv4
    int      ai_socktype; // e.g. SOCK_STREAM for TCP
    int      ai_protocol; // e.g. IPPROTO_TCP
    size_t   ai_addrlen;
    char     *ai_canonname;
    struct sockaddr *ai_addr; // point to sockaddr struct
    struct addrinfo *ai_next;
};
```

- Example

```c
hints.ai_family = AF_UNSPEC;  // don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
int status = getaddrinfo("www.cnn.com", "80", &hints, &result);
// result now points to a linked list of 1 or more addrinfos
// etc.
```
Client: Creating a Socket

- Creating a socket
  - `int socket(int domain, int type, int protocol)`
  - Returns a file descriptor (or handle) for the socket

- Domain: protocol family
  - `PF_INET` for IPv4
  - `PF_INET6` for IPv6

- Type: semantics of the communication
  - `SOCK_STREAM`: reliable byte stream (TCP)
  - `SOCK_DGRAM`: message-oriented service (UDP)

- Protocol: specific protocol
  - `UNSPEC`: unspecified
  - (`PF_INET` and `SOCK_STREAM` already implies TCP)

- Example
  ```c
  sockfd = socket( result->ai_family,
                  result->ai_socktype,
                  result->ai_protocol);
  ```
Client: Connecting Socket to the Server

- Client contacts the server to establish connection
  - Associate the socket with the server address/port
  - Acquire a local port number (assigned by the OS)
  - Request connection to server, who hopefully accepts
  - `connect` is **blocking**

- Establishing the connection
  - `int connect(int sockfd, struct sockaddr *server_address, socketlen_t addrlen)`
  - Args: socket descriptor, server address, and address size
  - Returns 0 on success, and -1 if an error occurs
  - E.g. `connect(sockfd, result->ai_addr, result->ai_addrlen);`
Client: Sending Data

- **Sending data**
  - `int send(int sockfd, void *msg, size_t len, int flags)`
  - Arguments: socket descriptor, pointer to buffer of data to send, and length of the buffer
  - Returns the number of bytes written, and -1 on error
  - `send` is **blocking**: return only after data is sent
  - Write short messages into a buffer and send once
Client: Receiving Data

• Receiving data
  
  – `int recv( int sockfd, void *buf, size_t len, int flags)`
  
  – Arguments: socket descriptor, pointer to buffer to place the data, size of the buffer
  
  – Returns the number of characters read (where 0 implies "end of file"), and -1 on error
  
  – Why do you need len? What happens if buf's size < len?
  
  – `recv` is **blocking**: return only after data is received
Byte Order

• Network byte order
  – Big Endian

• Host byte order
  – Big Endian (IBM mainframes, Sun SPARC) or Little Endian (x86)

• Functions to deal with this
  – htons() & htonl() (host to network short and long)
  – ntohs() & ntohl() (network to host short and long)

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

• Server creates a socket and binds address/port
  – Server creates a socket, just like the client does
  – Server associates the socket with the port number

• 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 sockfd, struct sockaddr *my_addr, 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
  – `int listen(int sockfd, int backlog)`
  – Arguments: socket descriptor and acceptable backlog
  – Returns a 0 on success, and -1 on error
  – Listen is **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
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: Cleaning House

• 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