Announcements

After class today: persons seeking project partners please gather at the front of the room.

Negotiating TCP options: “…both sides must send Window Scale options in their SYN segments to enable window scaling in either direction.”

Clients and Servers

<table>
<thead>
<tr>
<th>Clients</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Many</td>
<td>One</td>
</tr>
<tr>
<td>Initiate communication (active open)</td>
<td>Wait for communication request (listen)</td>
</tr>
<tr>
<td>Often make short request and then exit</td>
<td>Usually started at boot-up time, remain running for a long time</td>
</tr>
<tr>
<td>Use ephemeral ports, dynamically assignable</td>
<td>Use “well-known” (assigned) ports so it is easy to contact them</td>
</tr>
<tr>
<td>Implementation is usually simple</td>
<td>Implementation may be complex</td>
</tr>
</tbody>
</table>

Server Complexity

- Must be robust
- Must handle concurrent requests
- Must be secure (authenticate users and requests, etc.)
THE SOCKETS API

The Sockets Application Programming Interface (API)

- Goals: flexible / expressive, portable, easy to use
- Introduced in 1981 by Unix BSD 4.1
  - implemented as system calls
  - available on lots of platforms
- Types of service
  - datagram (UDP)
  - stream (TCP)
  - raw (plain IP, no transport layer)

For Full Documentation / Details

- See...
  - Comer, Volume III, Appendix I
  - Your OS’s online documentation (e.g., man connect)
  - Other web resources (see Links web page)

System Calls and Errors

- In general, systems calls return a negative number to indicate an error
  - servers generally log errors
  - clients generally provide some feedback to the user
- Whenever an error occurs, the value of the global variable errno is set
  - you can check errno for specific errors
  - you can use support functions to print out a text error message: perror(), strerror()
Constant and Function Definitions

- You’ll need to include these definitions for your programs to compile

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdarg.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
```

Data Structure: Socket Addresses

- IPv4 addresses = 32 bits
- Socket address = port + IP address

```c
struct in_addr {
    u_long s_addr; /* 32-bit address */
};

struct sockaddr_in {
    u_char sin_len; /* total length */
    short sin_family; /* AF_INET */
    u_short sin_port; /* port number */
    struct in_addr sin_addr; /* the IP address */
    char sin_zero[8]; /* unused, padding */
};
```

The Socket Abstraction

- Describes the state of a connection
  - also provides buffering
  - sockets are not bound to specific addresses at the time of creation
- Identified by small integer, the socket descriptor

```c
s = socket(PF_INET, type, ppe->p_proto);
```

Byte Ordering Definitions

- Byte ordering is a function of machine architecture
  - Intel: little-endian
  - Sparc, PowerPC: big-endian
  - Network order: big-endian
- With little-endian, you must do byte ordering conversions before writing / reading header fields
  - if never hurts to use, and it improves portability
  - how about the payload?
  - only necessary for multiple-byte words (16 bits, 32 bits)
**Byte-Order Transformations (cont’d)**

- Two versions: 16 bit and 32 bit

![Diagram of byte-order transformations]

**Byte Ordering Example**

- You store the value \(987,365,212_{10}\) in a 32-bit integer variable on a little-endian machine
- You send these 4 bytes across the network, *without* using network reordering (\(htonl()\) and \(ntohl()\))
- You read and print this 32-bit integer variable on a big-endian machine and get \(1,560,271,162_{10}\) – what happened?!

**Byte Ordering Example (cont’d)**

<table>
<thead>
<tr>
<th>LSBI</th>
<th>987,365,212</th>
<th>MSBI</th>
</tr>
</thead>
<tbody>
<tr>
<td>10111001</td>
<td>11111111</td>
<td>11011001</td>
</tr>
</tbody>
</table>

**Address Conversions**

- People work with dotted-decimal (character) representations of IP addresses or DNS names, machines work with 32-bit integer representations
  - \(\text{addr: network byte order, str: dotted decimal form}\)
  - \(\text{inet_addr}()\) to \(\text{inet_ntoa}()\)

**Ex.: Getting The IP Address**

```c
/* host either is a name ("semmes.csc.ncsu.edu"), */
/* or a dotted decimal IP address (e.g., "152.1.68.213") */
struct hostent *phe; /* pointer to host information */
if ( phe = gethostbyname(host) ) /* use the resolver */
    memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
else
    if ((sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE )
        (print error message here and exit);
```

**Ex.: Getting the Port Number**

```c
/* service either is a name (i.e., "echo"), or a port */
/* number (e.g., "7"), or transport is either "tcp" or "udp" */
struct servent *pse; /* pointer to service entry */
/* Map service name to port number */
if ( pse = getservbyname(service, transport) )
    sin.sin_port = pse->s_port;
else {
    sin.sin_port = htons((unsigned short) atoi(service));
    if (sin.sin_port == 0)
        (print error message here and exit);
}    ```
Ex.: Getting the Transport Protocol Type

```c
int type; /* transport is either "udp" or "tcp" */
struct protoent *ppe; /* pointer to protocol entry */

/* Map transport protocol name to protocol number */
if ( (ppe = getprotobyname(transport)) == 0) 
    (print error message here and exit) ;

/* Use protocol to choose a socket type */
if (strcmp(transport, "udp") == 0)
type = SOCK_DGRAM;
else
    type = SOCK_STREAM;
```

Obtaining Socket Addresses

- Newly created process may need to determine the addresses (local and remote) associated with a socket
  - only works with connected sockets
- Get the remote IP address and port number
  ```c
  getpeername(int s, struct sockaddr *peeraddr, int *peerlen)
  ```
- Get the local IP address and port number
  ```c
  getsockname(int s, struct sockaddr *localaddr, int *localism)
  ```

Types of Servers

<table>
<thead>
<tr>
<th>Connection-oriented: TCP</th>
<th>One connection at a time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal TCP</td>
<td>A few simple TCP applications</td>
</tr>
<tr>
<td>Connectionless: UDP</td>
<td>A few high-volume or long-duration UDP applications</td>
</tr>
<tr>
<td>Normal UDP</td>
<td>Normal UDP</td>
</tr>
</tbody>
</table>

Client-Server Interaction: UDP

<table>
<thead>
<tr>
<th>Step #</th>
<th>Server</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Create socket with <code>socket()</code></td>
<td>Create socket with <code>socket()</code></td>
</tr>
<tr>
<td>2a</td>
<td>Bind socket to local port and IP address with <code>bind()</code></td>
<td></td>
</tr>
<tr>
<td>2b</td>
<td>Bind socket to remote IP address and port with <code>bind()</code></td>
<td>(also chooses local port and binds socket to it)</td>
</tr>
</tbody>
</table>

Client-Server Interaction (cont’d)

<table>
<thead>
<tr>
<th>Step #</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Execute <code>recvfrom()</code>; block until data (a request) is received from a remote client, and record remote address</td>
</tr>
<tr>
<td>4</td>
<td>Send a request to the server with <code>write()</code></td>
</tr>
<tr>
<td>5</td>
<td>{<code>recvfrom()</code> unblocks, request can now be processed by the server} execute <code>read()</code> and block until response is sent by server</td>
</tr>
<tr>
<td>6</td>
<td>Send response to the client using <code>sendto()</code> and remote address</td>
</tr>
</tbody>
</table>
Client-Server Interaction (cont’d)

7 (read() unblocks, response can now be processed)

8 Execute close() and exit

9 Server goes back to step #3, waits for next request by a client

Creating A Socket with socket()

- Parameters
  - domain: PF_INET
  - type: SOCK_DGRAM, SOCK_STREAM, SOCK_RAW
  - protocol: usually = 0 (i.e., default for type)

- Example
  ```c
  s = socket(PF_INET, SOCK_STREAM, 0);
  ```

Result of socket()

<table>
<thead>
<tr>
<th>Server and Client sockets</th>
<th>Socket data structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Descriptor Table (one per process)</td>
<td>Family: PF_INET</td>
</tr>
<tr>
<td></td>
<td>Service: SOCK_STREAM</td>
</tr>
<tr>
<td></td>
<td>Local IP:</td>
</tr>
<tr>
<td></td>
<td>Local Port:</td>
</tr>
<tr>
<td></td>
<td>Remote IP:</td>
</tr>
<tr>
<td></td>
<td>Remote Port:</td>
</tr>
</tbody>
</table>

Binding to a Socket with bind()

- Used by servers to specify the well-known local port to use

- Optional for clients; system usually chooses an "available" local port

- Use INADDR_ANY to bind the socket to all of the machine’s interfaces (if multi-homed)
  ```c
  sin.sin_addr.s_addr = INADDR_ANY;
  if (bind(s, (struct sockaddr *) &sin, sizeof(sin)) < 0)
    (print error message here);
  ```

Result: bind()

<table>
<thead>
<tr>
<th>Server socket</th>
<th>Socket data structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Descriptor Table (one per process)</td>
<td>Family: PF_INET</td>
</tr>
<tr>
<td></td>
<td>Service: SOCK_STREAM</td>
</tr>
<tr>
<td></td>
<td>Local IP: INADDR_ANY</td>
</tr>
<tr>
<td></td>
<td>Local Port: 37</td>
</tr>
<tr>
<td></td>
<td>Remote IP:</td>
</tr>
<tr>
<td></td>
<td>Remote Port:</td>
</tr>
</tbody>
</table>

Connecting to Remote Endpoint with connect()

- Clients use connect() to specify a remote endpoint so other calls don’t have to
  - no packet exchange or connection establishment!

- Also binds to an available local port
  ```c
  if (connect(s, (struct sockaddr *) &sin, sizeof(sin)) < 0)
    (print error message here);
  ```
Result: `connect()`

- **Client socket**
  - Operating System
  - **Descriptor Table (one per process)**
    - 0
    - 1
    - 2
    - 3

- **Socket data structure**
  - Family: PF_INET
  - Service: SOCK_STREAM
  - Local IP: INADDR_ANY
  - Local Port: 1025
  - Remote IP: 192.168.15.35
  - Remote Port: 37

Steps #3-7: Sending and Receiving Datagrams

- Each function call sends or receives **one complete message** (datagram)
  - “Flags” parameter can modify action of the call
    - `recv()`, `send()`, `recvfrom()`, and `sendto()` can specify flags
    - `read()` and `write()` cannot

Receiving UDP Datagrams

- **`read(int s, char *buf, int buflen);`**
- **`recv(int s, char *buf, int buflen, int flags);`**
- **`recvfrom(int s, char *buf, int buflen, int flags, struct sockaddr *from, int fromlen);`**

  - Receives **up to `len` bytes** into buffer
    - returns number of bytes received
    - if `buf` is not large enough, any additional bytes in the datagram are discarded
  - `recvfrom()` records the remote endpoint address that sent the datagram if not connected

Sending UDP Datagrams

- **`write(int s, char* buf, int buflen);`**
- **`send(int s, char* buf, int buflen, int flags);`**
- **`sendto(int s, char *buf, int buflen, int flags, struct sockaddr *to, int tolen);`**

  - Sends up to `buflen` bytes
  - The return value indicates how much data was accepted by the O.S. for sending as a datagram
    - not how much data made it to the destination
    - there is no return code indicating destination got the data

Sending UDP Datagrams (cont’d)

- `sendto()` specifies the remote IP address / port number to which the data should be sent (if not connected)

Step #8: Closing UDP Sockets

- **`close(int s);`**

  - Releases the resources associated with a socket
  - Does **not** inform the remote endpoint that the socket is closed
    - i.e., there is no connection to terminate
**Summary**

1. The Sockets API is pretty much universal for network programming
2. Best advice
   1. Learn the functions
   2. Then develop and use a standard "template" for writing networked applications

_socket(), bind(), and listen() create a passive (unconnected) socket

UDP applications relatively easy to write: send a datagram, wait for a response

---

**Next Lecture**

- Sockets Programming Part II, and Server Design