Creating an Internet Client
CS 360 Internet Programming

Daniel Zappala

Brigham Young University
Computer Science Department
Clients request a service from a server using a protocol

- need API for creating and using TCP connections: socket API
- must define application protocol for requesting and receiving services
Processes and Sockets

- inter-process communication uses messages sent on sockets
- socket API defines
  - how to open, close, read, and write to socket
  - which transport protocol to use
  - various communication parameters
Socket API

- BSD Socket API - the dominant socket API on Linux, BSD, Windows
- Use the man pages!
  - shows C syntax, with include files
  - gives a description of how the system call works
  - lists return values
  - lists errors
  - lists other relevant man pages
- `man socket`
- the socket API is in section 2 of the manual, so you will need to specify the section if the same command exists in an earlier section: `man 2 bind`
Addresses and Ports

- to talk to a process on another machine, you need to identify it
  - IP address: identifies the machine
  - port number: identifies the socket the process is using
IPv4 address identifies an interface/link on a host or router
- 32 bits
- dotted-decimal notation: each part is 8 bits
Ports

- identifier for a socket on which a process is listening
  - 16 bits
  - a process may listen on more than one socket; each must be on a separate port

- Operating systems (such as Linux) designate some ports as privileged, meaning only the superuser can listen on them (typically ports less than 1024)

- To help find common servers, the IANA designates well-known ports for many protocols
  - HTTP: 80
  - SMTP: 25
  - SSH: 22
  - NTP: 123
Creating a Client

1. create a `socket()`
2. `connect()` to a server
3. `send()` and `recv()` data
4. `close()` the socket
Creating a Socket

```c
int socket(int domain, int type, int protocol);
```

3  `domain = PF_INET`
4  `type = SOCK_STREAM for TCP, SOCK_DGRAM for UDP`
5  `protocol = 0`

- on success returns a socket descriptor
Connecting to a Server

```c
1 int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
2
3 sockfd = socket you created
4 serv_addr = pointer to socket address structure
5 socklen_t = length of socket address
```

- connects to a server
- uses the socket address structure to pass an IP address and port
- on success returns zero
- on error returns -1 and sets errno
Sending and Receiving Data

- usually the client sends a request to the server and the server sends a reply
- socket operations are similar to reading from and writing to a file
  - a socket descriptor acts like a file handle
  - sending ~ writing
  - receiving ~ reading
Send and Receive Syntax

1. `ssize_t send(int s, const void *buf, size_t len, int flags);`
2. `ssize_t recv(int s, void *buf, size_t len, int flags);`

- `s` = socket
- `buf` = pointer to buffer
- `len` = size of buffer in bytes
- **on success returns the number of bytes actually sent or received**
  - if it is less than what you expected, then you must repeat the system call until all data is sent or received
- `recv()` will return 0 if the socket has been closed
- **on error returns -1 and sets errno**
- see man pages for more advanced options
Closing a Socket

```c
#include <unistd.h>

int close(int fd);
```

- `fd = socket`
- `on success returns zero`
- `on error returns -1 and sets errno`
- `releases the socket file descriptor so it can be re-used!`
to connect a socket, you must supply a socket address structure that includes the IP address and port of the server you are connecting to

1. use DNS to convert host name to IP address
2. initialize a socket address structure
Domain Name System

- people like to use names for computers (www.byu.edu), but computers need to use numbers (128.187.22.132)
- the Domain Name System (DNS) is a distributed database providing this service
  - a program send a query a local name server
  - the local name server contacts other servers as needed
- many DNS services
  - host name to IP address translation
  - host aliasing (canonical name versus alias names)
  - lookup mail server for a host
  - load distribution - can provide a set of IP addresses for one canonical name

Demonstration: DNS services
Using DNS

1. \texttt{struct hostent *gethostbyname(const char *name);}  

- \texttt{name} = host name  
- on success returns a hostent structure  
- on error returns NULL

1. \texttt{struct hostent {  
2. char *h_name; /* official name of host */  
3. char **h_aliases; /* alias list */  
4. int h_addrtype; /* host address type */  
5. int h_length; /* length of address */  
6. char **h_addr_list; /* list of addresses */  
7. }}

\textit{See DNS example code on class web site.}
Generic Socket Address Structure

```c
    struct sockaddr {
        sa_family_t  sa_family;    // address family
        char         sa_data[14];  // address
    }
```

- used to represent a generic connection between processes
- potentially provides access to many different addressing standards
- address family can be
  - `AF_UNIX` : local UNIX socket
  - `AF_INET` : Internet socket
IPv4 Socket Address Structure

```c
struct sockaddr_in {
    sa_family_t    sin_family;    // address family
    u_int16_t      sin_port;      // port
    struct in_addr sin_addr;     // Internet address
    char           sin_zero[8];   // unused
};
```

- can cast IPv4 socket into a generic socket
- port is 2 bytes, address is 4 bytes, zero is 8 bytes = 14 bytes
- IPv4 address structure:

```c
struct in_addr {
    u_int32_t    s_addr;        // address
};
```
Network Byte Order

- You must store the port and address in network byte order (most significant byte sent first)
- Provides interoperability among Internet hosts
- Use `htons()` for the port and `inet_aton`, `inet_addr()` or `inet_makeaddr()` for the address

```c
1 struct sockaddr_in server;
2 server.sin_port = htons(80);
3 if (!inet_aton(ipaddress, &server.sin_addr))
4     printf("inet_addr() conversion error\n");
```
See client-server example code on class web site.