Socket Programming
CS 360 Internet Programming

Daniel Zappala

Brigham Young University
Computer Science Department
Clients and Servers

- clients request a service from a server using a protocol
- need an API for sending and receiving data
- need an abstraction for the Internet
  - a reliable connection: TCP
  - an unreliable message service: UDP
- these are provided by the **BSD socket API**
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 man
- 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
Client API

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

```c
1 int socket(int domain, int type, int protocol);
2
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 $\sim$ writing
  - receiving $\sim$ 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!
### Example

- *See client-a socket code on class web site.*
Socket Addresses

- 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
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:
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");
```
Example

- See *client-a socket code on class web site*. 
Server API

1. create a **socket**
2. **bind**: associate the socket with an address and port
3. **listen**: convert socket to listen for incoming connections
4. **accept**: accept an incoming client connection
5. **send**: send data
6. **recv**: receive data
1. `int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);`

4. `sockfd = socket you created`

5. `my_addr = pointer to socket address structure`

6. `addrlen = length of socket address`

- associates an address with a socket
- uses the socket address structure to pass an IP address and port
- on success returns zero, -1 and errno otherwise
Listening on the Socket

```c
int listen(int sockfd, int backlog);
```

- `sockfd` = socket you created and bound
- `backlog` = maximum waiting connections

- converts a socket to a passive socket (one that accepts connections rather than connecting)
- backlog is the maximum number of waiting connections the kernel should hold in a queue
- maximum value in Linux is 128
- on success returns zero, -1 and errno otherwise
Accepting a Connection

```c
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- sockfd = socket you created and bound
- addr = pointer to empty socket address structure
- addrlen = length of empty socket address structure

- accept a connection from a client; gets the next connection waiting in the queue
- if there are no pending connections, the process sleeps assuming this is a blocking socket (this is the default)
- on success returns a new socket descriptor
- on success, accept() fills in the address of the client
Example

See server-a socket code on class web site.
Demultiplexing

- when a new connection arrives
  - delivered to a queue for the listening socket
  - `accept()` creates a new client socket

- when data for an existing connection arrives
  - delivered to existing socket
  - identified by
    (source IP address, source port, destination IP address, destination port)
an application protocol reads and write messages, but TCP and the socket API use a byte stream

- when a client calls recv() there is no guarantee that it will get an entire message
- the application has to designate where a message starts and ends and then parse the byte stream looking for messages

- two options for reading a message
  1. variable length: read until a sentinel (end-of-message marker)
  2. known length: read a length field and then read the listed number of bytes

- you will need to write a read-until-sentinel() and read-fixed-length() methods for your socket programming labs

Try client-a with server-b in the socket example code. Now try client-b with server-b in the socket example code.
Sending and Receiving Properly

- results of \texttt{send()} or \texttt{recv()} call
  1. less than zero bytes, \texttt{errno == EINTR}
     - the system call was interrupted – try again
  2. less than zero bytes, any other error
     - fatal error – try to recover gracefully
  3. zero bytes
     - the socket is closed – try to recover gracefully
  4. positive bytes
     - the return value is the number of bytes sent or received

- use a \texttt{send()} or \texttt{recv()} loop!
Example

• See client-c and server-c socket code on class web site.