UNIX Socket Programming
Introduction to Sockets

• What are Sockets?
  – End-point of inter-process communication.
  – An interface through which processes can send / receive information

• The Client-Server model
  – Most inter-process communication uses client-server model
  – The Client process connects to the Server process,
    • To make a request for information/services own by the Server.
  – Once the connection is established between them,
    • They can start sending / receiving information.
Introduction to Sockets

• What exactly creates a Socket?
  – <IP address, Port #> tuple

• What makes a connection?
  – Source<IP addr, Port #>
  – Destination <IP addr, Port #>

• Source socket + destination socket pair uniquely identifies a connection.
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
Socket Addressing
(IP Address + Port Number)

A woman contacts a salesman with Company Phone Number + Extension Number
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. Server **Accepts** connection requests from clients

4. Further communication is specific to application
A typical algorithm for TCP app.

• **Algorithm for TCP client**
  – Find the IP address and port number of server
  – Create a TCP socket
  – Connect the socket to server (Server must be up and listening for new requests)
  – Send/ receive data with server using the socket
  – Close the connection

• **Algorithm for TCP server**
  – Find the IP address and port number of server
  – Create a TCP server socket
  – Bind the server socket to server IP and Port number *(this is the port to which clients will connect)*
  – Listen and accept a new connection from client
  – Send/ receive data with client using the client socket
  – Close the connection with client
Concurrent Communications

1. connect

3. Communication with send/recv

2. fork

1. connect

Communication with send/recv
Creating a socket

```c
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

```c
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

```
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 */
};
```
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
Connect Example

```c
int sd;
struct sockaddr_in sa;
sd = socket(AF_INET, SOCK_STREAM, 0);
sa.sin_family = AF_INET;
sa.sin_port = htons(5100);
//htons() converts host-byte-order into network-byte-order
sa.sin_addr.s_addr = inet_addr("128.101.34.78");
if (connect(sd, (struct sockaddr *) &sa, sizeof(sa)) != -1)
...
```
Byte Ordering

• **Integers:**
  – **Little Endian:**
    • least significant byte first
    • E.g., DEC, Intel
  – **Big Endian**
    • most significant byte first
    • E.g., Sun, SGI, HP

• **Network Byte Order**
  – **Big Endian**

• **Byte Order Conversion**
  – \( m = \text{n}t\text{o}hl(m) \): network-to-host byte order, 32bit
  – \( m = \text{h}t\text{o}nl(m) \): host-to-network byte order
  – \text{ntohs, htonl}: short(16bit)
Big Endian vs. Little Endian

32-bit Integer

0xaabbccddd
Address: X

BIG ENDIAN

X
X+1
X+2
X+3

0xaabbccddd

0xaabbccddd

LITTLE ENDIAN

X
X+1
X+2
X+3

Address: X

32-bit Integer

MSB

LSB

MSB

LSB
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 information
- **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
Typical Server Structure

sockid = socket()

bind()

listen()

newsockid = accept()

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

child process

parent 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
    • TCP layer makes IP layer send multiple IP datagrams of TCP
      Maximum Segment Size (MSS).

• Connectionless unreliable datagram
  – SOCK_DGRAM based on UDP
  – Message boundaries are preserved
  – Each sendto corresponds to one recvfrom
    • Each sendto generates an IP datagram.
Multiplexing with 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
    • 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

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 */
}
FD_ZERO(&readset);

FD_SET(0, &readset);

FD_SET(4, &readset);

result = select(5, &readset, NULL, NULL, NULL);

if(FD_ISSET(0, &readset)) { // true since fd 0 is set to 1
...
}

if(FD_ISSET(4, &readset)) { // true since fd 4 is set to 1
...
}
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
Example: Echo Client/Server
Example: echo-server

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int s, t;
    struct sockaddr_in sin;
    char msg[80];
    int sinlen;
    if (argc < 2) { printf("%s port\n", argv[0]); return -1; }
    if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket"); return -1;
    }
}
sin.sin_family = AF_INET;
sin.sin_port = htons(atoi(argv[1]));
sin.sin_addr.s_addr = INADDR_ANY;

if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
    perror("bind"); return -1;
}

if (listen(s, 5) < 0) { perror("listen"); return -1; }
sinlen = sizeof(sin);
if ((t = accept(s, (struct sockaddr *)&sin, &sinlen)) < 0) {
    perror("accept"); return -1;
}

printf("From %s:%d.\n", inet_ntoa(sin.sin_addr), ntohs(sin.sin_port));
if (read(t, msg, sizeof(msg)) < 0) { perror("read"); return -1; }
if (write(t, msg, sizeof(msg)) < 0) { perror("write"); return -1; }
if (close(t) < 0) { perror("close"); return -1; }
if (close(s) < 0) { perror("close"); return -1; }

return 0;
Example : echo-client

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    int s;
    struct sockaddr_in sin;
    char msg[80] = "Hello World !";
    int n;

    if (argc < 3) {
        printf("%s host port\n", argv[0]);
        return -1;
    }
```
if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket"); return -1;
}

sin.sin_family = AF_INET;
sin.sin_port = htons(atoi(argv[2]));
sin.sin_addr.s_addr = inet_addr(argv[1]);

if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
    perror("connect"); return -1;
}

if (write(s, msg, strlen(msg)+1) < 0) {
    perror("write"); return -1;
}

if ((n = read(s, msg, sizeof(msg))) < 0) {
    perror("read"); return -1;
}

printf("%d bytes: %s\n", n, msg);
if (close(s) < 0) { perror("close"); return -1; }
return 0;