Socket Programming

15-441 Computer Networks, Spring 2010
Your TAs
Lecture Today

• Motivation for sockets
• What’s in a socket?
• Working with socket
• Concurrent network applications
• Project 1
Why Socket?

• How can I program a network application?
  – Share data
  – Send messages
  – Finish course projects...

• IPC - Interprocess Communication
Identify the Destination

- **Addressing**
  - IP address
  - hostname (resolve to IP address via DNS)

- **Multiplexing**
  - port
Sockets

• How to use sockets
  – Setup socket
    • Where is the remote machine (IP address, hostname)
    • What service gets the data (port)
  – Send and Receive
    • Designed just like any other I/O in unix
    • send -- write
    • recv -- read
  – Close the socket
Overview

Client

socket
connect
write
read
close

Server

socket
bind
listen
accept
read
write
read
close

open_clientfd
Connection request
open_listenfd
EOF

Client / Server Session
Step 1 – Setup Socket

• Both client and server need to setup the socket
  – \textit{int socket(int domain, int type, int protocol);}  

• \textit{domain}
  – AF_INET -- IPv4 (AF_INET6 for IPv6)

• \textit{type}
  – SOCK_STREAM -- TCP
  – SOCK_DGRAM -- UDP

• \textit{protocol}
  – 0

• For example,
  – \textit{int sockfd = socket(AF_INET, SOCK_STREAM, 0);}
Step 2 (Server) - Binding

- Only server need to bind
  - int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);

- sockfd
  - file descriptor socket() returned

- my_addr
  - struct sockaddr_in for IPv4
  - cast (struct sockaddr_in*) to (struct sockaddr*)

```
struct sockaddr_in {
  short sin_family;   // e.g. AF_INET
  unsigned short sin_port;  // e.g. htons(3490)
  struct in_addr sin_addr;  // see struct in_addr, below
  char sin_zero[8];   // zero this if you want to
};
struct in_addr {
  unsigned long s_addr;  // load with inet_aton()
};
```
What is that Cast?

- bind() takes in protocol-independent (struct sockaddr*)

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

- C’s polymorphism
- There are structs for IPv6, etc.
Step 2 (Server) - Binding contd.

- `addrlen`
  - size of the sockaddr_in

```c
struct sockaddr_in saddr;
int sockfd;
unsigned short port = 80;

if((sockfd=socket(AF_INET, SOCK_STREAM, 0) < 0) {  // from back a couple slides
  printf("Error creating socket\n");
  ...
}

memset(&saddr, '\0', sizeof(saddr));  // zero structure out
saddr.sin_family = AF_INET;  // match the socket() call
saddr.sin_addr.s_addr = htonl(INADDR_ANY);  // bind to any local address
saddr.sin_port = htons(port);  // specify port to listen on

if((bind(sockfd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0) {  // bind!
  printf("Error binding\n");
  ...
}
```
What is htonl(), htons()?

• Byte ordering
  – Network order is big-endian
  – Host order can be big- or little-endian
    • x86 is little-endian
    • SPARC is big-endian

• Conversion
  – htons(), htonl(): host to network short/long
  – ntohs(), ntohl(): network order to host short/long

• What need to be converted?
  – Addresses
  – Port
  – etc.
Step 3 (Server) - Listen

• **Now we can listen**
  – `int listen(int sockfd, int backlog);`

• **sockfd**
  – again, file descriptor socket() returned

• **backlog**
  – number of pending connections to queue

• **For example,**
  – `listen(sockfd, 5);`
Step 4 (Server) - Accept

- **Server must explicitly accept incoming connections**
  - `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)`
- **sockfd**
  - again... file descriptor socket() returned
- **addr**
  - pointer to store client address, (struct sockaddr_in *) cast to (struct sockaddr *)
- **addrlen**
  - pointer to store the returned size of addr, should be sizeof (*addr)
- For example
  - `int isock=accept(sockfd, (struct sockaddr_in *) &caddr, &clen);`
struct sockaddr_in saddr, caddr;
int sockfd, clen, isock;
unsigned short port = 80;

if((sockfd=socket(AF_INET, SOCK_STREAM, 0) < 0) { // from back a couple slides
    printf("Error creating socket\n");
    ...
}

memset(&saddr, '\0', sizeof(saddr));    // zero structure out
saddr.sin_family = AF_INET;       // match the socket() call
saddr.sin_addr.s_addr = htonl(INADDR_ANY); // bind to any local address
saddr.sin_port = htons(port);    // specify port to listen on

if((bind(sockfd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0) { // bind!
    printf("Error binding\n");
    ...
}

if(listen(sockfd, 5) < 0) {  // listen for incoming connections
    printf("Error listening\n");
    ...
}

clen=sizeof(caddr)
if((isock=accept(sockfd, (struct sockaddr *) &caddr, &clen)) < 0) { // accept one
    printf("Error accepting\n");
    ...
}
What about client?

• Client need not bind, listen, and accept
• **All client need to do is to connect**
  
  – `int connect(int sockfd, const struct sockaddr *saddr, socklen_t addrlen);`

• For example,
  
  – `connect(sockfd, (struct sockaddr *) &saddr, sizeof (saddr));`
Domain Name System (DNS)

- What if I want to send data to “www.slashdot.org”?
  - DNS: Conceptually, DNS is a database collection of host entries

```
struct hostent {
    char    *h_name;    // official hostname
    char    **h_aliases;   // vector of alternative hostnames
    int     h_addrtype;    // address type, e.g. AF_INET
    int     h_length;    // length of address in bytes, e.g. 4 for IPv4
    char    **h_addr_list; // vector of addresses
    char    *h_addr;     // first host address, synonym for h_addr_list[0]
};
```

- hostname -> IP address
  - `struct hostent *gethostbyname(const char *name);`

- IP address -> hostname
  - `struct hostent *gethostbyaddr(const char *addr, int len, int type);`
Put Client Together

```
struct sockaddr_in saddr;
struct hostent *h;
int sockfd, connfd;
unsigned short port = 80;

if((sockfd=socket(AF_INET, SOCK_STREAM, 0) < 0) { // from back a couple slides
  printf("Error creating socket\n");
    ...
}

if((h=gethostbyname("www.slashdot.org")) == NULL) { // Lookup the hostname
  printf("Unknown host\n");
    ...
}

memset(&saddr, '\0', sizeof(saddr));    // zero structure out
saddr.sin_family = AF_INET;       // match the socket() call
memcpy((char *) &saddr.sin_addr.s_addr, h->h_addr_list[0], h->h_length); // copy the address
saddr.sin_port = htons(port);    // specify port to connect to

if((connfd=connect(sockfd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0) { // connect!
  printf("Cannot connect\n");
    ...
}
```
We Are Connected

• Server accepting connections and client connecting to servers
• Send and receive data
  – ssize_t read(int fd, void *buf, size_t len);
  – ssize_t write(int fd, const void *buf, size_t len);
• For example,
  – read(sockfd, buffer, sizeof(buffer));
  – write(sockfd, “hey\n”, strlen(“hey\n”));
TCP Framing

• TCP does NOT guarantee message boundaries
  – IRC commands are terminated by a newline
  – But you may not get one at the end of read(), e.g.
    • One Send “Hello\n”
      • Multiple Receives “He”, “ll\n”
  – If you don’t get the entire line from one read(), use a buffer
Revisited

Client / Server Session

Client
- socket
- connect
- write
- read
- close

Server
- socket
- bind
- listen
- accept
- read
- write
- close

open_clientfd
open_listenfd
Connection request
EOF
Close the Socket

• Don’t forget to close the socket descriptor, like a file
  – `int close(int sockfd);`

• Now server can loop around and accept a new connection when the old one finishes

• What’s wrong here?
Server Flaw

Client 1

- call connect
- ret connect
- call fgets

User goes out to lunch

Client 1 blocks waiting for user to type in data

Server

- call accept
- ret accept
- call read

Server blocks waiting for data from Client 1

Client 2

- call connect

Client 2 blocks waiting to complete its connection request until after lunch!

Taken from D. Murray, R. Bryant, and G. Langale 15-441/213 slides
Concurrent Servers

Client 1

- call connect
- ret connect
- call fgets

User goes out to lunch

Client 1 blocks waiting for user to type in data

Server

- call accept
- ret accept
- call read (don’t block)
- call accept
- ret accept

Client 2

- call connect
- ret connect
- call fgets
- write
- call read
- write
- end read
- close
- close

Taken from D. Murray, R. Bryant, and G. Langale 15-441/213 slides
Concurrency

• Threading
  – Easier to understand
  – Race conditions increase complexity

• Select()
  – Explicit control flows, no race conditions
  – Explicit control more complicated

• There is no clear winner, but you MUST use select()...
What is select()?

• Monitor multiple descriptors
• How does it work?
  – Setup sets of sockets to monitor
  – select(): blocking until something happens
  – “Something” could be
    • Incoming connection: accept()
    • Clients sending data: read()
    • Pending data to send: write()
    • Timeout
Concurrency – Step 1

• Allowing address reuse

```c
int sock, opts=1;
sock = socket(...);  // To give you an idea of where the new code goes
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &opts, sizeof(opts));
```

• Then we set the sockets to be non-blocking

```c
if((opts = fcntl(sock, F_GETFL)) < 0) { // Get current options
    printf("Error...
");
    ...
}
opts = (opts | O_NONBLOCK);  // Don't clobber your old settings
if(fcntl(sock, F_SETFL, opts) < 0) {
    printf("Error...
");
    ...
}
bind(...);  // To again give you an idea where the new code goes
```
Concurrency – Step 2

- **Monitor sockets with select()**
  - `int select(int maxfd, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, const struct timespec *timeout);`
- **maxfd**
  - max file descriptor + 1
- **fd_set**: bit vector with FD_SETSIZE bits
  - `readfds`: bit vector of read descriptors to monitor
  - `writefds`: bit vector of write descriptors to monitor
  - `exceptfds`: set to NULL
- **timeout**
  - how long to wait without activity before returning
What about bit vectors?

• `void FD_ZERO(fd_set *fdset);`
  – clear out all bits

• `void FD_SET(int fd, fd_set *fdset);`
  – set one bit

• `void FD_CLR(int fd, fd_set *fdset);`
  – clear one bit

• `int FD_ISSET(int fd, fd_set *fdset);`
  – test whether fd bit is set
The Server

// socket() call and non-blocking code is above this point

if((bind(sockfd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0) { // bind!
    printf("Error binding\n");
    ...
}

if(listen(sockfd, 5) < 0) {  // listen for incoming connections
    printf("Error listening\n");
    ...
}

clen=sizeof(caddr);

// Setup pool.read_set with an FD_ZERO() and FD_SET() for
// your server socket file descriptor. (whatever socket() returned)

while(1) {
    pool.ready_set = pool.read_set; // Save the current state
    pool.nready = select(pool.maxfd+1, &pool.ready_set, &pool.write_set, NULL, NULL);

    if(FD_ISSET(sockfd, &pool.ready_set)) { // Check if there is an incoming conn
        isock=accept(sockfd, (struct sockaddr *) &caddr, &clen); // accept it
        add_client(isock, &pool); // add the client by the incoming socket fd
    }

    check_clients(&pool); // check if any data needs to be sent/received from clients
}

...
What is pool?

typedef struct { /* represents a pool of connected descriptors */
    int maxfd;     /* largest descriptor in read_set */
    fd_set read_set; /* set of all active read descriptors */
    fd_set write_set; /* set of all active read descriptors */
    fd_set ready_set; /* subset of descriptors ready for reading */
    int nready;      /* number of ready descriptors from select */
    int maxi;        /* highwater index into client array */
    int clientfd[FD_SETSIZE]; /* set of active descriptors */
    rio_t clientrio[FD_SETSIZE]; /* set of active read buffers */
    ...  // ADD WHAT WOULD BE HELPFUL FOR PROJECT1
} pool;
What about checking clients?

• The main loop only tests for incoming connections
  – There are other reasons the server wakes up
  – Clients are sending data, pending data to write to buffer, clients closing connections, etc.

• Store all client file descriptors
  – in pool

• Keep the while(1) loop thin
  – Delegate to functions

• Come up with your own design
Summary

- Sockets
  - socket setup
  - I/O
  - close
- Client: socket() \rightarrow connect() \rightarrow I/O \rightarrow close()
- Server: socket() \rightarrow bind() \rightarrow listen() \rightarrow accept() \rightarrow I/O \rightarrow close()

- DNS
  - gethostbyname()
- Concurrency
  - select()
- Bit vector operations
  - fd_set, FD_ZERO(), FD_SET(), FD_CLR(), FD_ISSET()
About Project 1

• Standalone IRC server
  – Checkpoint 1: subversion and Makefile
    • Check in a Makefile and source code
    • Makefile can build executable named `sircd`
    • No server functions necessary
  – Checkpoint 2: echo server
    • Use `select()` to handle multiple clients
Suggestions

• Start early!
  – Work ahead of checkpoints
• Read the man pages
• Email Kaushik (kaushik AT cs DOT cmu DOT edu) if you didn’t get a svn username and password
Reference for Socket Programming

• [http://www.lowtek.com/sockets/](http://www.lowtek.com/sockets/)

• Feel free to share your reference on the bboard!