Introduction to Network Programming using C/C++

Target Audience

- Prior knowledge of C programming is expected
- The lecture is for introducing the data structures and APIs used in network programming (beginner level)
- Sample code shown are for unix environment

Would be giving Introduction about...

- Parsing command line parameters
- Address structures used by network programming APIs
- Address Conversion/Resolution functions
- Byte Order Conversion
- Socket types and creating a socket
- Making TCP connections
- UDP data transfer
- Multicast Send/Receive
- Sending and Receiving data
- Blocking and Non-Blocking sockets
- Event Driven Programming
- I/O Multiplexing using select()
- Error Checking

Parsing Command line parameters

Function: int **getopt** (int argc, char **argv, const char *options)

Defined in library: unistd.h

Example: (./ProgramName -n xyz.hut.fi -s -p 5345)

```
int opterr = 0, c = 0;
while ((c = getopt (argc, argv, "h:sp:")) != -1) {
        switch(c) {
                case 'n': ResolveHostName(optarg); break;
                case 's': sFlag = 1; break;
                case 'p': GotPortNumber(optarg); break;
                case '?':
                        HandleError(); // prints Usage Instructions
        }
}
```

Socket Address Structures

```
(i)  struct sockaddr_in {
        short sin_family;           // (Address family AF_INET)
        unsigned short   sin_port;  // Port Number
        struct in_addr   sin_addr;// Expanded below
        char    sin_zero[8];        // holds zeroes
};
        struct in_addr {
            unsigned long s_addr;
           /* contains a unique number for each IP address.
              The output of inet_aton() is stored here */
        };
(ii)struct sockaddr {
    short int sa_family;
    char sa_data[14];
};
```

Socket Address Structures contd...1

- Both sockaddr and sockaddr_in structures are of same length.
- Socket functions like bind(), recvfrom(), sendto() etc use sockaddr structure.
- The normal practice is to fill the **stuct sockaddr_in** and cast the pointer to **struct sockaddr** while socket operartions.

Example: (Note: Since it is example – return codes are not checked)

```
struct sockaddr_in ServAddr;
ServAddr.sin_family = AF_INET;
ServAddr.sin_port = htons(5345);
inet_aton("130.233.x.y", &ServAddr.sin_addr); // (refer next slide for inet_aton)

int sd = socket(PF_INET, SOCK_STREAM, 0);
bind(sd, (struct sockaddr *)&ServAddr, sizeof(struct sockaddr));
```

Address Conversion Functions

- Ipv4 Conversion functions: Converts dotted IP address format to a format understandable by the socket APIs and vice versa.

```
in_addr_t inet_aton
        (const char *IP_Address, struct in_addr *addr);
char * inet_ntoa(struct in_addr in);
```

- Similar Conversion functions for Ipv6 are

inet_pton() and inet_ntop()

Socket Address Structures contd...2

```
struct hostent {
    char        *h_name;        // Official name of the host
     char       **h_aliases;    // Alternative names
     int        h_addrtype;     // Address Type (AF_INET)
     int        h_length;       // Length of each address
     char       **h_addr_list;  // Address List
     char       *h_addr;        // h_addr_list[0]
};
```

-gethostbyname() and gethostbyaddr() uses this address structure

- gethostbyname:
  struct hostent * gethostbyname (const char *Host_Name)

- gethostbyaddr: (addr is a pointer to struct in_addr)
  struct hostent * gethostbyaddr
  (const char *addr, size_t length, int format)

Name/IP Addr. resolution functions

- Functions explained here are used for performing HostName to IP address and vice-versa mappings
- These functions are defined in file netdb.h
- They use /etc/hosts or a name server for resolving the address

gethostbyname() Example:

```
char *HostName = "xyz.hut.fi"; // or an IP address 130.233.x.y
struct hostent *hp = gethostbyname(HostName);
```

gethostbyaddr() Example:

```
/* Assume that the struct sockaddr_in ServAddr is already filled with proper values   (refer slide 6) */
struct hostent *hp = gethostbyaddr(
                (char *)&ServAddr.sin_addr.s_addr,
                 sizeof(ServAddr.sin_addr.s_addr),
                AF_INET)
```

Byte order conversion

- Network and Host byte order
  - All data in the network are sent in 'Big Endian' format
  - But different systems use different byte orders
    (i.e., different ways of storing bytes in memory)
  - Calling these functions are necessary when setting the address parameters that are passed to socket APIs
  - Example: **unsigned short var = 255; // 0x00FF**

    Little Endian-> **FF 00** (Host Byte Order)

    Big Endian-> **00 FF** (Network Byte Order)
- Functions used for this conversion purpose

  htons() and ntohs() -> for 16 bit variable conversion

  htonl() and ntohs() -> for 32 bit variable conversion

Socket types

- Sockets are the entry and exits through which different process communicate
- Different communication method require different socket types
  - SOCK_STREAM for TCP
  - SOCK_DGRAM for UDP
  - SOCK_RAW for sending RAW IP packets
  - SOCK_PACKET for sending Link Layer frames
- Example: **sd = socket(AF_INET, SOCK_DGRAM, 0);**

/* the last argument specifies the protocol, it is normally kept as '0'. some special case where it is used is, when creating SOCKET_RAW */

- **'sd'** is called a socket descriptor (the concept is similar to the FILE descriptor which we are familiar with)
- At this step(after socket() function is called) the socket is not related to any particular IP address(and port number)

Making TCP connections

Server Mode

- Make a socket with SOCK_STREAM option (refer previous slide)
- Bind the socket to a address (particular IP and Port number)

```
int rc = bind(sd, (struct sockaddr *)(&ServAddr), sizeof(struct sockaddr))
```

- Listen on the socket to accept new connections

```
int rc = listen(int sd, int backlog);
/* backlog ->specifies the number of connections that has to be queued by
   the kernel */
```

- Accept the connected clients

```
new_sd = accept (int sd, struct sockaddr *peer, socklen_t *peerlen);
/* new_sd -> It is a new socket descriptor
   The originalsocket descriptor is still used for listening to incoming
  connection requests */
```

Making TCP connections contd...

Client Mode

- Make a socket with SOCK_STREAM option
- Fill in the server address in the struct sockaddr_in ServAddr;
- Connect to the server by giving the server address

  connect (int sd, struct sockaddr *ServAddr, socklen_t len)

  For Blocking I/O - Function call completes only when the connection is established or if a time-out occurs without response or when ICMP error messages indicate failure (e.g., destination unreachable)
- Closing connections
  - **shutdown (int sd, int mode)**

/* 0: no further sending, 1: no further reception, 2: neither sending nor receiving */

  - **close(sd)** to clean up – data loss possible

UDP data transfer

Sending and Receiving data over UDP:

- Make a socket with SOCK_DGRAM option
- Bind the socket to a particular IP address and Port Number
- Now the socket can be used for both sending and receiving data

(The send and recv functions are described in the next slide)

Note for UDP: If you intend to receive data only from a particular IP address and port number, then you need to verify the source address of the packet immediately after receiving the datagram.

Multicasting

- Multicast is same(in coding aspects) as unicast UDP
- Create a datagram socket and fill the Multicast address in the Target address structure

```
struct sockaddr_in targetAddr;
memset(&targetAddr,0,sizeof(targetAddr));
targetAddr.sin_family=AF_INET;
targetAddr.sin_addr.s_addr=inet_addr(MULT_GROUP);
targetAddr.sin_port=htons(MULT_PORT);

sendto(sd,mBuf,sizeof(mBuf),0,
    (struct sockaddr *) &targetAddr, sizeof(targetAddr));
```

Multicast Reception

- Steps to receive Multicast packets
  - Create a datagram socket
  - Bind the socket to the Port no. in which Multicast data need to be received
  - Fill in the structure

```
struct ip_mreq {
  struct in_addr imr_multiaddr; // IP Multicast address of the group
  struct in_addr imr_interface; // Local IP Address of the interface
}
```

  - Use setsockopt() to ADD Multicast Membership
    - configures the socket to the specific Multicast reception
  - recv() on the bound sd can receive Multicast packets
- At Exit: Again setsockopt() is used to DROP membership

Multicast Reception contd..2

- **MULTICAST JOIN:**

```
struct ip_mreq mRecv;
mRecv.imr_multiaddr.s_addr=inet_addr(MULT_GROUP);
mRecv.imr_interface.s_addr=htonl(INADDR_ANY);
setsockopt(sd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
        &mRecv, sizeof(mRecv));
recv(sd, ...) // Receive packets
```

- **MULTICAST LEAVE:**

```
setsockopt(sd, IPPROTO_IP, IP_DROP_MEMBERSHIP,
        &mRecv, sizeof(mRecv));
```

Sending and Receiving data

Sending

-Connection-oriented (TCP)

write (int sd, char *buffer, size_t length);

send (int sd, char *buffer, size_t length, int flags);

-Connectionless (UDP)

sendto (int sd, char *buffer, size_t length, int flags,
struct sockaddr *target, socklen_t addrlen);

Receiving

- Connection-oriented (TCP)

read (int sd, char *buffer, size_t length);

recv (int sd, char *buffer, size_t length, int flags);

- Connectionless (UDP)

recvfrom (int sd, char *buffer, size_t length, int flags,
struct sockaddr *target, socklen_t addrlen)

(If you observe here, only udp requires us to specify the address value each time we do transfer data)

Blocking and Non-blocking sockets

- When we call call recvfrom(), the system call checks if any data is available at the kernel buffer. If so, it would return with the data.
- What if no data is available when recvfrom() is called?
    - Default Action: It blocks on the call, till it gets the data.
    - But if we do not want our program to block in this situation, then the socket need to be set as non-blocking.
    - In non-blocking mode, the recvfrom() returns with error message EWOULDBLOCK (indicating that no data available to be read)
- function **int fcntl(int sfd, int cmd, int flags)**
    - Using the flags variable, socket can be made non-blocking.
- In the assignment point of view, we recommend to use blocking mode, which is the default mode.

Event Driven Programming

- Assignments of this course requires concurrency feature i.e., your program need to listen on a particular port number and accept many incoming connections. All established connections has to be handled concurrently.

- For such a kind of requirement, the concurrency can be provided **without** using threads.

- Event driven programming approach does not use threads. Lets see an example of how they achieve concurrency without threads.

In the assignment point of view: We do not recommend threads.

Note: You are free to chose the method, using which you provide the functionality to handle multiple requests. There are discussions which detail on which method provides better throughput. But here, we leave it to your choice.

Event driven programming contd..

Using threads approach:

```
loop {
    if  ( listen_for_incoming_connection() ) {
        accept_new_connection();
        create_new_thread_to_process_request();
    } // Goes back listening for incoming connection
}
```

Using event driven approach:

- Maintain a list of channels(socket descriptors) to listen on to for events
- Poll for events on the maintained list of channels (using **select()**)
- Maintain state for each accepted connection.
- Process events based on its type and its stored state values.

I/O Multiplexing (polling for events)

select (int number_fds, fd_set *read_fds, fd_set *write_fds, fd_set *exception_fds, struct timeval *timeout)

- **fd_set** is a variable type used by select. It is used store the values of the socket descriptors that we need to listen on.

- There are macro functions available to process **fd_set** variables.

void FD_SET (int sockdes, fd_set *target_set)

- sets the **sockdes** in the **target_set**

FD_CLR (int filedes, fd_set *set)

- resets the **sockdes** in the **target_set**

FD_ISSET (int filedes, const fd_set *set)

- checks if **sockdes** is set in **target_set**

select contd...

- Register the socket descriptors in the fd_set
- Call **select()**
  - **Error** if (fatal) terminate;
    else if (repairable) repeat_select_call;
    // Ex – if error is EINTR
  - **Time-out**
    - the time value is specified using the struct timeval
    - NULL pointer represents no time-out
      (blocks till one of the socket descriptors report for action)
    - if timeval is set to {0, 0} -> then it returns immediately (polling)
  - **Success**
    - Determine the active descriptors and handle events

select() example

```
rc_select = select (max_sd + 1, &working_fd_set, NULL, NULL, &select_timeout);
/* Check to see if the select call failed. */
if (rc_select < 0) {
  perror("select() failed");
  check error number and act accordingly
}
/* Check to see if the 'n' second time out expired.       */
if (rc_select == 0) {
  fprintf(stderr, "\n select() timed out. \n");
  return -1;
}
.....
   /* Check to see if there is a incoming connection request or data to be read */
   if (FD_ISSET(sd, &working_fd_set)) {
                .......
```

Checking for errors

- #include<errno.h> uses a variable errno, that are used by functions to report error.
- function calls(socket(), bind(), send(), recv() etc) set errno value that can be used to spot the error quickly.
- Here we see an example

```
rc = bind (int sd, struct sockaddr *ServAddr, socklen_t length);
if (rc < 0) {
        perror("bind failed:" ); // prints the errno value in a string format
        Call_Exit_Routine();
}
```

- When this code(bind()) gets executed with wrong function parameters, the possible output values are

bind failed: socket already has an address

(you cannot call bind for second time on the same socket)

Miscellaneous..

For students interested in creating a svn repository in the university unix machines, please refer the below link.

http://goblin.tkk.fi/c++/tutorials/svn.html

Questions ?