Introduction to Network Programming using C/C++

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
- UDP data transfer
- Sending and Receiving data
- Blocking and Non-Blocking sockets
- I/O Multiplexing using select()
- Protocol FSM
- Packet Pacing
- Scheduling Events

Parsing Command line parameters

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

Defined in library: unistd.h

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

```
int opterr = 0, c = 0;
while ((c = getopt (argc, argv, "h:sp:")) != -1) {
        switch(c) {
                case 'h': 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 APIs bind(), recvfrom(), sendto() use sockaddr structure.
- The normal practice is to fill the **stuct sockaddr_in** and cast its pointer to **struct sockaddr** while calling the functions

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 to a representation understandable by the socket APIs and vice versa.

```
int 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)

UDP data transfer

Sending and Receiving data over UDP:

- Make a socket with SOCK_DGRAM option
- Bind the socket to a 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.

Sending and Receiving data

Sending data over UDP

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

Receiving data over UDP (bind() necessary)

bind (int sd, struct sockaddr *target, socklen_t len);

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

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. (to reduce implementation complexity, and ease of debugging)

I/O Multiplexing (event driven)

Possible Socket Events: READ_AVAILABLE, WRITE_READY

select (int max_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
  - **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)

Implementing a Protocol - FSM

- Protocol operation can be modelled as Finite State Machine
- Protocol have a set of States
  - Example: enum protoState {WAITING, READY, BLOCKED}
- State of the Protocol Changes depending on an event
  - Event Can be arrival of a message(data, ack), timeouts, memory constraints etc
  - Events can trigger change in state of the protocol, can trigger new events(like sending ACK)
  - Processing of events depends on the current state of the protocol
- Using FSM, deadlocks can be identified

Packet Pacing

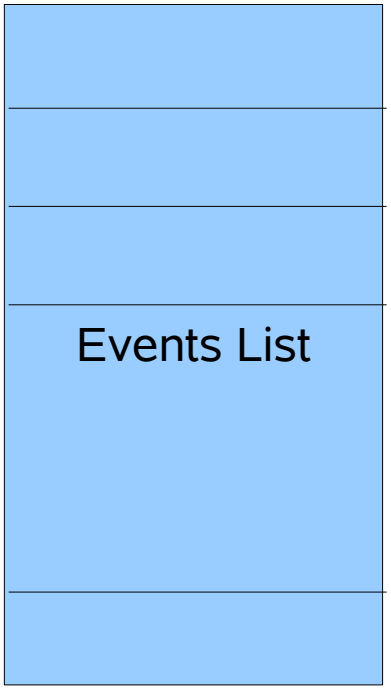
- To achieve a target bit rate, need to send packets in regular intervals
- Total Payload Size = Your own protocol header size + 8 bytes UDP + 20 bytes IPv4 + Payload Size
- Inter Packet Interval(IPI)=Total Payload Size/Target BitRate
- Create a timeout event with timeout value as IPI
  - Handler that handles this event shall reschedule itself at expiry
  - While rescheduling, IPI may need to be recalculated if the state variables indicate change in target bit rate

Scheduling Events

- Protocol Operation involves many timeouts
  - Timeouts for packet pacing
  - Timeouts for triggerring ACK
  - Timeouts that signals lost connection
  - and many more depending on your protocol features
- Every Timeout is an scheduled event, represented as
  - <timeOfExpiry, handlerFunction>

  At the expiration of the timer, the corresponding handler function is invoked
- Maintain a sorted list of timeouts(sorted on timeOfExpiry)

Scheduling Events Contd..1



eventId addingEvent(Time, callBackFunction)

void cancelEvent(eventId)
[At times you need to cancel a scheduled Event

void modifyEvent(eventId, time, callBackFunction)

At the expiry of an event, the registered callBackFunction shall be executed. If the callBackFunction needs a set of args, then it can also be specified during the process of adding an Event.
Ex: addingEvent(time, callBackFunction, **args**)

Scheduling Events Contd..2

- A timeout value can be specified with the select() function
- This timeout value specified in the select() is taken from the EventList(refer prev slide)
- While waiting to execute the next scheduled event, the select() might return with READ_READY Event on a particular socket descriptor.
  - The message(event) read from the socket can possibly insert new event or remove/cancel an already scheduled event in the EventList

Questions ?