Introduction to Network Programming using C/C++

Slides mostly prepared by Joerg Ott (TKK) and Olaf Bergmann (Uni Bremen TZI)
Would be giving brief introduction on...

- Parsing Command line
- Socket Related Address Structures
- Host Name / IP Address resolution
- Socket Creation
- Making TCP and UDP Connection
- Sending and Receiving Data
- Multicasting
- Multiplexing I/O
- Handling Timeouts
- Packet Pacing
- Random Number Generators
- Suggestions & Hints for the Assignment
Parse Command Line

```c
int getopt(cnt, argv, optarg, optstring)

int oc;
while( (oc=getopt(argc, argv,"a:bi:sl:D:t:")) != -1) {
    switch(oc) {
        case 'a' : addAddress(optarg); break;
        case 'b' : usage(); exit(0);
        case 'i' : addInterface(optarg); break;
        case 's' : summary = true; break;
        case 'l' : dumplen = GetInt(optarg); break;
        case 't' : controlAddress(optarg); break;
        case 'D' : duration = GetInt(optarg); break;
        default :
            opterr(oc);
    }
}
```
Address Structures

- struct sockaddr_in {
  - uint8_t sin_len; /* length of structure (16) */
  - sa_family_t sin_family; /* AF_INET */
  - in_port_t sin_port; /* 16-bit TCP or UDP port number */
  - struct in_addr sin_addr; /* 32-bit IPv4 address */
  - char sin_zero[8];
};

- struct in_addr {
  - in_addr_t s_addr; /* 32-bit IPv4 address */
};

- struct sockaddr {
  - uint8_t sa_len;
  - sa_family_t sa_family; /* address family: AF_xxx value */
  - char sa_data[14]; /* protocol-specific address */
};
Address Structures Contd...

- `bind()`, `recvfrom()` and `sendto()` function uses sockaddr structure
- A normal practice is to fill the stuct sockaddr_in and cast the pointer to struct sockaddr while socket operations

```c
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() returns the resolved address in struct hostent format. A hostname may have multiple interfaces, so hostent structure is designed to hold the multiple addresses of the resolved hostname
Address Conversion functions (1)

**Ipv4 Conversion:**

- `in_addr_t inet_addr (char *buffer)`
- `in_addr_t inet_aton (char *buffer)`
- `char * inet_ntoa (in_addr_t ipaddr)`

**For Ipv6 Conversion:**

- `int inet_pton(int af, const char *src, void *dst)`
  - `dst: in_addr or in6_addr`

- `const char *inet_ntop`
  - `(int af, const void *src, char *dst, size_t)`
  - `src: in_addr bzw. in6_addr`
  - `char dst[INET_ADDRSTRLEN] bzw. char
dst[INET6_ADDRSTRLEN]`
Conversion Functions (2)

Network vs. Host Byte Order:

All data in the network is sent as “Big Endian”
Conversion into little Endian representation required for Intel
Example: unsigned short var = 255; (0x00FF)
Little Endian: FF 00 (Host Byte Order)
Big Endian: 00 FF (Network Byte Order)

netshort = htonl (hostshort)
netlong = htonl (hostlong)
hostshort = ntohs (netshort)
hostlong = ntohl (netlong)
Socket Types

- Socket Descriptor: similar to file i/o or stdin/stdout
- Each socket descriptor represents a connection or a particular IP and Port address

- Supports different types of communications, u.a.
  - SOCK_STREAM: TCP
  - SOCK_DGRAM: UDP
  - SOCK_RAW: Raw IP
  - SOCK_PACKET: Link-Layer-Frames
Socket Creation

```c
int socket(domain, type, proto)
int bind(sd, addr, addrlen)
```

```c
int createSocket(const sockaddr_in &addr) {
    int sd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sd < 0) return -1;

    int yes = 1;
    setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, (char *)&yes, sizeof yes);
    fcntl(sd, F_SETFL, O_NONBLOCK);
    if (bind(sd, (struct sockaddr *)&addr, sizeof(struct sockaddr)) < 0) {
        std::cerr << strerror(errno) << std::endl;
        return -1;
    }
    return sd;
}
```
Creating UDP and TCP connections

**UDP:**
- Create a socket with `SOCK_DGRAM`
- Bind the socket to a address (particular IP and port number)
- `Ex- bind (int sd, struct sockaddr *, socklen_t len);`
- Now the socket can be used for send and receive operations

**TCP:**
- Create a socket with `SOCK_STREAM`
- Bind the socket to a address (particular IP and port number)
- If program need to accept any connection request, then listen on the socket
- `Listen()` - allows to specify the number of backlogs of connection requests that can be buffered
Connections (TCP) contd..

- Connecting to a listening end
  - `connect (int sd, struct sockaddr *target, socklen_t len);`
  - Function call only complete when the connection is established, if a timeout occurs without response (may be several minutes), or when ICMP error messages indicate failure (e.g., destination unreachable)

- Accepting an incoming connection (cannot reject anyway:)
  - `new_sd = accept (int sd, struct sockaddr *peer, socklen_t *peerlen);`
  - Creates a new socket descriptor for the new connection
  - The original one (sd) continues to be used for accepting further connections

- Closing a connection
  - `shutdown (int sd, int mode)`
  - 0: no further sending, 1: no further reception, 2: neither sending nor receiving
  - `close(sd)` to clean up – beware of data loss!
Sending Data

- **Connection-oriented (TCP)**
  - `write (int sd, char *buffer, size_t length);`
  - `writev (int sd, struct iovec *vector, int count);`
    - List of buffers, each with pointer to memory and 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)`
  - `sendmsg (int sd, struct msghdr *msg, int flags)`
    - Target address
    - Pointer to the memory containing the data
    - Control information
Receiving Data

- **Connection-oriented (TCP)**
  - `read` (int sd, char *buffer, size_t length);
  - `recv` (int sd, char *buffer, size_t length, int flags)
    - List of buffers, each with pointer to memory and length

- **Connectionless (UDP)**
  - `recvfrom` (int sd, char *buffer, size_t length, int flags, struct sockaddr *target, socklen_t addrlen)
  - `recvmsg` (int sd, struct msghdr *msg, int flags)
    - Sender address
    - Pointer to the data
    - Control information
Further Functions

- **getpeername** (int sd, struct sockaddr *peer, size_t *len)
  - Obtain the address of the communicating peer
- **getsockname** (int sd, struct sockaddr *local, size_t *len)
  - Obtain the address of the local socket (e.g., if dynamically assigned)

- **Modify socket parameters**
  - **getsockopt** (int sd, int level, int option_id, char *value, size_t length)
  - **setsockopt** (int sd, int level, int option_id, char *value, size_t length)
  - Examples:
    - Buffer size, TTL, Type-of-Service, TCP-Keepalive, SO_LINGER, ...
    - `fcntl (int sd, int cmd [, long arg] [, ...]);`
  - Non-blocking I/O
Multicast reception

- Multicast JOIN
  ```c
  setsockopt (sd, IPPROTO_IP, IP_ADD_MEMBERSHIP, 
              struct ip_mreq *mreq, sizeof(ip_mreq));
  struct ip_mreq {
    struct in_addr imr_multiaddr;     /* IP multicast address of group */
    struct in_addr imr_interface;     /* local IP address of interface */
  };
  ```

- Multicast-LEAVE
  ```c
  setsockopt (sd, IPPROTO_IP, IP_DROP_MEMBERSHIP, struct ip_mreq *mreq, sizeof (ip_mreq));
  ```

- Optional: Allow repeated use of an address (needed for multicasting)
  ```c
  char one = 1;
  setsockopt (sd, SOL_SOCKET, SO_REUSEADDR, &one, sizeof (char))
  ```
I/O Multiplexing (select)

`int select(maxfdset, read, write, ext, timer)`

- socket descriptors specified in the file descriptor set (FDSET)
- Determine earliest timeout
- Call `select()`
- Error?
  - Fatal - Terminate
  - Repairable (e.g. interrupted system call) - repeat
- Timeout?
  - Timer handling; use `struct timeval { ... }` to specify (sec, usec) pair
  - NULL pointer == blocking (no timeout), (0, 0) == polling
- Success
  - Determine active file descriptors and handle events
fd_set Makros used by select

fd_set base_set working_set;
FD_ZERO (&working_set);
FD_SET (fd, &base_set);
.
.
.
if (FD_ISSET(fd, &working_set))
  . . .
Select() example

```c
rc_select = select (max_sd + 1, &working_set, NULL, NULL, &select_timeout);
/* Check to see if the select call failed. */
if (rc_select < 0)
{
    perror("select() failed");
    check errno and act accordingly
}
/* Check to see if the 'n' minute 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 */
if (FD_ISSET(sd, &working_set))
{
    ........
    ........
```
I/O Multiplexing (poll)

```c
int poll(pollfd,n_fd,timeout)
```

- **struct pollfd**
  ```c
  struct pollfd {
  int fd;           // file descriptor
  int events;      // events to watch for
  int revents;     // occurred events
  }
  ```

- **Poll events**:
  - POLLIN          input pending
  - POLLOUT         socket writable (only needed with non-blocking i/o)
  - POLLHUP, POLLERR

- **Timeout is specified in milliseconds**
  - -1 == no timeout, 0 == return immediately (perform real polling)

- **Handling otherwise identical to select()**
Timeouts

- Protocols use many timeouts
  - Some Examples of timeouts are, (i) timeouts used for packet pacing, (ii) retransmission timeouts
  - An occurrence of an event may change (set/reset/cancel) the timeout variables
  - Must be implemented efficiently

- `select()` and `poll()` allow you to specify a timeout value
  - In `poll()`, timeout is specified in milliseconds
  - and `select()` provides microseconds resolution (uses `struct timeval`)

- Keep an ordered list of all your timeouts
  - Store absolute time for the timeout
  - Event this timeout is about (a timeout event may trigger a change in STATE of the protocol)

- Before calling `select/poll`
  - Determine current time (`gettimeofday()`)
  - Determine first timeout in list and calculate delta
    (if timeout has already passed initiate handling right away)
  - Parameterize `poll/select()` with the delta
Timeouts ...contd

Example:
Timeout 200ms

```c
struct timeval tv, delta, now;

/* some event occurs -> calculate absolute time in tv */
gettimeofday (&tv, NULL);
tv.tv_usec += 200*1000;
if (tv.tv_usec >= 1000000) {
    tv.tv_usec -= 1000000;
    tv.tv_sec++;
}

/* ... many other activities -> back in mainloop */
gettimeofday (&now, NULL);
delta.tv_usec = tv.tv_usec - now.tv_usec;
delta.tv_sec = tv.tv_sec - now.tv_sec;
if (delta.tv_usec < 0) {
    delta.tv_usec += 1000000;
    delta.tv_sec--;
}
if (delta.tv_sec < 0) {
    /* timeout has also passed -> handle now */
}
switch (n = select (... , ..., ..., ..., ..., &delta) { 
    ...
}
Packet pacing

- To achieve a target bit rate, need to send packets in regular intervals
- Calculate your target packet interval from the packet size…
  - Your own header + 8 bytes UDP + 20 bytes IPv4 + 1024 bytes payload
  - and the target bit rate on the command line
- Use a recurring timer for transmission
  - Important: calculate your transmission interval based upon a single initial absolute time value
    - E.g. calculate your initial transmission time based upon gettimeofday()
    - Always add your constant interval to the previous timeout value without calling gettimeofday() again for this purpose
  - Do not do regular calculations
    - This will lead to underutilization as it does not account for local processing time
Random number generators

- **int rand() and void srand(unsigned int seed)** ISO C
  - srand sets the seed value of the generating function
  - Call to rand() generates a random number between 0 and RAND_MAX (using GNU C Library)
  - RAND_MAX: 2147483647 (largest signed integer representable using 32 bits)

- **long int random() and void srandom(unsigned int seed)** BSD
  - Their working is very similar to the ISO C functions

- **double drand48() and void srand48(long int seed)** SVID
  - Uses a state of 48 bits of data, provides better randomness than ISO and BSD functions
  - Call to drand48() generates a value in the range of 0.0 to 1.0 (exclusive)
  - srand48() can initialize only the 32 bits of the state data, but the function unsigned short * seed48(unsigned short seed[3]) can be used initialize all the 48 bits of state data.
Beware of threads

- If your coding language allows you to avoid them
  - Will save you hassle (and overhead) in synchronizing access to internal data structures

- Instead
  - Maintain your own state explicitly in some data structure
  - Remember what to do next
    - E.g., send data at a certain time, wait for a response, etc.
  - “Register” all socket descriptors for your mainloop
  - “Register” all your timeouts
  - Process incoming events for all contexts one by one
Hints (1)

- Transport address(es) to receive data on
  - socket (SOCK_DGRAM, AF_INET, ...)
  - Create and bind an individual UDP socket for every address
  - Remember host vs. network byte order

- Generation of artificial packet loss
  - Write your own small lossy_sendto (...)
  - Use drand48() instead of rand() or random()

```c
double p_loss = ...;

lossy_sendto (int sd, void *msg, size_t len, ...) {
    if (drand48 () > p_loss) {
        return sendto (sd, msg, len, ...);
    }
    return len;
}
```
Hints (2)

- **Timer handling**
  - `gettimeofday(2)` yield detailed system clock reading as (sec, usec) pair
  - If you work with timeout, calculate its absolute time
  - In the mainloop, determine the time to wait based upon the current time
    - This result is what you feed into `poll()` or `select()`
    - Note that both use completely different time formats
  - If `poll()/select()` returns 0, a timeout has occurred

- **DO NOT USE SIGNALS FOR TIMING**
  - Such as done by `alarm()`
  - This may just cause system call interruptions that you do not want or need
  - Better to stay in control all the time
Hints (3)

► Signals
- You may need to catch at least SIGINT: signal (SIGINT, signalhandler);
  - In this case, you would just set a global variable and return (terminate = 1;)
  - Need to check the variable regularly even if no packets arrive
- Will cause interrupted system calls (errno == EINTR)
  - Need to check for this also in your main loop and behave accordingly

► File access
- Regular i/o operation (open/close/read/write, fopen/fclose/fread/fwrite)
- MS Windows: you may need O_BINARY to avoid end of line conversion
- Use fstat () to obtain file attributes (including file size)