Introduction to Network Programming

Assignment 1: uft

Slides partly prepared by Olaf Bergmann (Uni Bremen TZI)

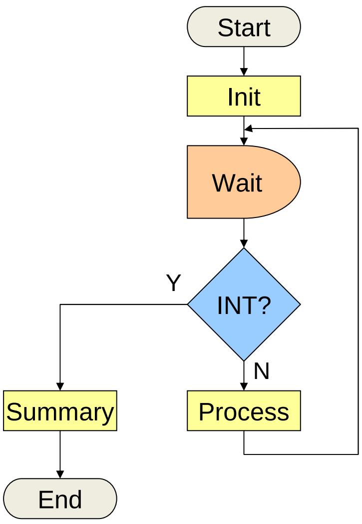
Starting Point

- IDE
  - Unix/Linux available in the department
  - Alternative: cygwin (winsock vs. BSD)
  - Also: MacOS (which is Unix), MS Windows
  - (the three items above: GNU gcc, make, gdb, ...)
  - Programming language: your choice
    - Examples and hints will be given in C/C++
- Information sources
  - Today's slides and exercise
  - Details on the web page
  - man, info, Google
  - Newsgroup
  - Send mail (if everything else has failed)

The Goals

- Workable software
  - Remember that you will need to build upon this later
  - Compiled and tested on the department workstations (Unix/Linux), on your laptop, or some other system accessible from the department via ssh
  - Learning: how to get there
  - Functionality: to actually arrive at a working solution
- Documentation
  - Motivation + protocol documentation: explicitly as text or PDF
  - Code: Inline
  - Shows that you understood the problem and the solutions
  - Helps you to remember what you were thinking today in two months from now
  - Helps us to understand what you meant to do
  - → There should be no “wrong” solutions (only malfunctioning ones)
- Working with development tools
  - make, gcc, gdb, cvs/svn, (autoconf) ...

Program Structure



Initialization
- Parse the command line & arguments
- Resolve hostname
- Bind sockets, join multicast groups (if any)
- Manage signal handling

Main loop
- Manage socket descriptors (there will be many)
- Read data
- Create output
- Signal and failure handling

Cleanup
- Close all descriptors
- Leave multicast groups (if any)
- Free memory

Parse Command Line

`int getopt(cnt,argv,optstring)`

```
int oc;
while( (oc=getopt(argc,argv,"a:hi:sl:D:t:")) != EOF)
{
  switch(oc) {
    case 'a' : addAddress(optarg); break;
    case 'h' : usage(); exit(0);
    case 'i' : addInterface(optarg); break;
    case 's' : summary = true; break;
    case 'l' : dumplen = strtol(optarg,NULL,10); break;
    case 't' : controlAddress(optarg); break;
    case 'D' : duration = strtol(optarg,NULL,10); break;
    default :
      opterr(oc);
  }
}
```


Resolve hostname

- Transform a symbolic name into a protocol-specific address
  ⇨ Attention: different address formats and lengths
- APIs
  - **`gethost*(), inet_aton(), inet_ntoa()`**
  - **`getaddrinfo(), inet_pton(), inet_ntop()`**

⇨ old

The Old Stuff

```
gethostname (char *name_buffer, int buffer_length)
struct hostent *gethostbyname (char *namestr)
struct hostent *gethostbyaddr (struct sockaddr *, size_t, int);

struct hostent {
   char       *h_name;
   char       **h_aliases;
   int        h_addrtype;
   int        h_length;
   char       **h_addr_list;
#define h_addr      h_addrlist [0]
};

struct hostent *gethostent ();
endhostent ();
```


getaddrinfo

```
int getaddrinfo(host,server,hints,result)
```

```
struct addrinfo {
     int              ai_flags;      /* AI_PASSIVE, AI_CANONNAME,
                                        AI_NUMERICHOST */
     int              ai_family;     /* PF_UNSPEC */
     int              ai_socktype;   /* SOCK_xxx */
     int              ai_protocol;   /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
     size_t           ai_addrlen;    /* length of ai_addr */
     char             *ai_canonname; /* canonical name for nodename */
     struct sockaddr  *ai_addr;      /* binary address */
     struct addrinfo  *ai_next;      /* next structure in linked list */
};

void freeaddrinfo(struct addrinfo *res);
const char *gai_strerror(int errcode);
```

Conversion functions (1)

Dotted decimal notation: aaa.bbb.ccc.ddd (IPv4 only)

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

aaa.bbb.ccc.ddd (IPv4), aaaa:bbbb:cccc:dddd:eeee:ffff:gggg:hhhh (IPv6)

```
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]

HELSINKI UNIVERSITY OF TECHNOLOGY
NETWORKING LABORATORY

Conversion Functions (2)

Network vs. Host Byte Order

All data in the network is sent as “Big Endian”
Conversion into local representation may be required

- needed on “Little-Endian” (LSB-first) architectures such as Intel
- is a no-op on MSB-first, but should **always** be done for portability

```
netshort  = htons (hostshort)      16-bit value
netlong   = htonl (hostlong)       32-bit value
hostshort = ntohs (netshort)       16-bit value
hostlong  = ntohl (netlong)        32-bit value
```

BSD Socket Interface

- The BSD mechanism for Inter-Process Communication (IPC)
- Transparency between local and remote communications
- Socket Descriptor: feels like file i/o or stdin/stdout

- Support for different address families (some 30 in socket.h)
  - (Named) Pipes (e.g., AF_UNIX), ...
  - Internet Protocols (AF_INET, AF_INET6)
  - Other
- Crucial for the spreading of IP in the 1980s!
- Supports different types of communications, u.a.
  - SOCK_STREAM: TCP   SOCK_DGRAM: UDP
  - SOCK_RAW: Raw IP   SOCK_PACKET: Link-Layer-Frames

Socket Creation

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

```
int createSocket(const sockaddr_in &addr)
{
    int sd=socket(PF_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,reinterpret_cast<const sockaddr *>(&addr),sizeof addr)<0) {
        std::cerr << strerror(errno) << std::endl;
        return -1;
    }
    return sd;
}
```

Socket domain
PF_INET, PF_INET6
Socket type
SOCK_STREAM, SOCK_DGRAM, ...
Protocol
0 (any), 6 (tcp), 17 (udp)

Address Structures

- Identification of a peer by means of IP address, port number, and protocol

```
struct sockaddr_in {
        sa_family_t    sin_family;
        in_port_t      sin_port;
        struct in_addr sin_addr;
    };
```

```
struct sockaddr_in6 {
        sa_family_t     sin6_family;
        in_port_t       sin6_port;
        uint32_t        sin6_flowinfo;
        struct in6_addr sin6_addr;
    };
```

IPv4 address (historically motivated, cumbersome)

```
struct in_addr {
    in_addr_t    s_addr;
};
```

IPv6 address (abbreviated)

```
struct in6_addr {
        uint8_t     u6_addr8[16];
#define s6_addr in6_u.u6_addr8
};
```


Passive Waiting

- Data reception (UDP), accepting incoming connections (TCP)
- **`bind (int sd, struct sockaddr *, socklen_t len);`**
- UDP: done
- TCP: enable connection setup from others
  - **`listen (int sd, in backlog);`**
  - Permits <backlog> pending connection setup requests in the kernel
- `setsockopt ()` and `ioctl ()` to set further parameters
  - Buffer size, Type-of-Service, TTL, multicast addresses, ...

Connections (TCP)

- `connect (int sd, struct sockaddr *target, socklen_t len);`
  - Creates (synchronously) a connection
  - Function call returns only when the connection is established, if a timeout occurs without response (may be several minutes), or *possibly* when ICMP error messages indicate failure (e.g., destination unreachable)
  - Option: TCP_NODELAY for asynchronous connection setup
- 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!

HELSINKI UNIVERSITY OF TECHNOLOGY
NETWORKING LABORATORY

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)`
    - May be used for out-of-band data
- 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);`**
  - **`readv (int sd, struct iovec *vector, int count);`**
    - List of buffers, each with pointer to memory and length
  - **`recv (int sd, char *buffer, size_t length, int flags)`**
    - May be used for out-of-band data

- 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 (useful 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, ...
  - **`ioctl (int sd, int request, ...);`**
  - **`fcntl (int sd, int cmd [, long arg] [, ...]);`**
    - E.g., to control whether I/O is non-blocking

Multicast reception

- Multicast JOIN

```
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
  - `setsockopt (sd, IPPROTO_IP, IP_DROP_MEMBERSHIP, struct ip_mreq *mreq, sizeof (ip_mreq));`
- Optional: Allow repeated use of an address (needed for multicasting)
  - `char one = 1;`
  - `setsockopt (sd, SOL_SOCKET, SO_REUSEADDR, &one, sizeof (char))`

I/O Multiplexing (select)

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

- Calculate file descriptor sets (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

```
fd_set wfdset;
FD_ZERO (&wfdset);
FD_SET (fd, &wfdset);
  .
  .
  .
if (FD_ISSET(fd, &wfdset))
   . . .
```


I/O Multiplexing (poll)

`int poll(pollfd,n_fd,timeout)`

-
```
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 (1)

- Protocols use many timeouts
  - Will be set, reset, and canceled frequently
  - Must be implemented efficiently
- select () and poll () allow you to specify one timeout
  - poll () in milliseconds
  - select () microseconds via struct timeval
- Keep an ordered list of all your timeouts
  - Store absolute time for the timeout
  - Pointer to the context (e.g., local protocol state of the "connection")
  - Event this timeout is about
- 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 (2)

Example:
Timeout 200ms

```
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 getttimeofday ()
    - 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

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

UDP file transfer: uft

- "Reliable" transfer of a file from one endpoint to another
- Two client modes of operation
  - Initiate a transfer from a server: send a request and then wait for data
  - Initiate a transfer to a server: send data
- Two server modes of operation
  - Wait for incoming requests for files from a client
  - Wait for incoming files from a client
- Distinguished by means of command line options

- File transmission shall take place in chunks of 1024 bytes
- File identification to be conveyed (i.e., the file name)
- File size to be included
- File checking information (e.g., a checksum)
- Other information conceivable

- Support "simulated" packet loss


```
uft [-a|-s] [-p <port>] [-l <lossrate>]
uft <host> [-p <port>] [-l <lossrate>] –b <bitrate> [-t|-r] <file>
```

-a:	server mode: accept incoming files from any host
-s:	server mode: accept requests for files to send from any host
-t:	client mode: send a file to the target host
-r:	client mode: request transmission of a file from a host
<host>	the host to send to or request from (hostname or IPv4 address)
-p:	specify the port number to use (use a default if not given)
-b:	transmission bitrate for the file (gross transmission rate)
<file>	the name of the file to send or request

Further options may be useful; up to you.
Remember to do report errors (locally and across the network) as needed.
You may want to do something useful if the user aborts either process (Ctrl-C).

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 (...)

```
double p_loss = ...;

lossy_sendto (int sd, void *msg, size_t len, ...) {
    if ((double) rand () / (double) MAXRANDNUMBER > 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);
    - This may occur at any point in time, so you may want to postpone processing to the main loop (probably not needed in our simple example)
    - 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 eol conversion
  - Use fstat () to obtain file attributes (including file size)

Hints (4)

```
/* command line processing goes here */

if ((s = socket (AF_INET, SOCK_DGRAM, 0)) == -1) {
   perror ("cannot create socket");
   exit (-1);
}

listen_addr.sin_family      = AF_INET;
listen_addr.sin_addr.s_addr = INADDR_ANY;
listen_addr.sin_port        = htons (listen_port);
if (bind (s, (struct sockaddr *) &listen_addr,
    sizeof (listen_addr)) == -1) {
    perror ("cannot bind to address");
    exit (-1);
}

... recvfrom (s, ...); ...
... sendto (s, ...); ...
```

Hints (5)

```
n_fd           = 1;
fds [0].fd     = s;
fds [0].events = POLLIN;
fds [0].revents= 0;

for (;;) {

for (i = 0; i < n_fd; i++)
    fds [i].events = POLLIN;

switch (poll (fds, n_fd, 600000)) {
    case -1:
        perror ("poll failed");
        sleep (1);
        break;
    case 0:
        fprintf (stderr, "Timeout...\n");
        break;

    default:
        for (i = 0; i < n_fd; i++) {
          if (fds [i].revents & POLLIN) {
              if (i == 0) {
                  /* process events on socket fds [i].fd */
              }
}}}
```