# Introduction to Network Programming

# Assignment 1: udpspy

Slides partly prepared by Olaf Bergmann (Uni Bremen TZI)

## Starting Point

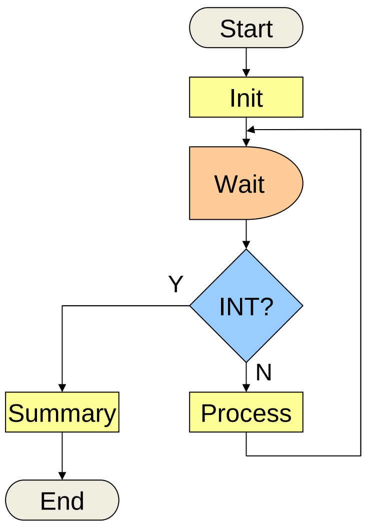
- IDE
  - Unix/Linux available in the department
  - Alternative: cygwin (winsock vs. BSD)

  } GNU gcc, make, ...
- Information sources
  - Today's slides and exercise
  - Details on the web page
  - man, info, Google
  - 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)
  - Learning: how to get there
  - Functionality: to actually arrive at a working solution
- Documentation
  - 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, 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]

## Conversion Functions (2)

Network vs. Host Byte Order

All data in the network is sent as “Big Endian”

Conversion into local representation required (Intel)

(depends on the CPU architecture but should always be done for portability)

```
netshort  = htons (hostshort)
netlong   = htonl (hostlong)
hostshort = ntohs (netshort)
hostlong  = ntohl (netlong)
```

## 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 (z.B. 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)
```

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

```
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;
}
```

## 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-Adresse (historisch motiviert umständlich)

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

IPv6-Adresse (hier verkürzt dargestellt)

```
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 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)
  - 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!

## 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 (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, ...
  - **`ioctl (int sd, int request, ...);`**
  - **`fcntl (int sd, int cmd [, long arg] [, ...]);`**
    - Non-blocking I/O

## 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()

## udpspy

- Receives UDP packets from a specified transport address (command line)
- Works for unicast and multicast addresses (IPv4, optionally IPv6)
- Virtually “any number” of addresses
- Short and long form for showing data packets
- If a control connections is present, dump the data to this connection (do not worry about the connection being too slow at this point)
- Terminating the program with Ctrl-C (SIGINT) will cause it to dump a summary of the packets received so far.
- Alternatively, a time period can be specified on the command line

- Test mode: Program sends data provided by the user to a specified transport address (to talk to your own client)
- Test sender **`pc27`**: **`226.226.226.226/62226`**, one packet per second containing some 120 bytes of text and binary data, and a sequence number (both in text and binary)

```
udpspy -a <addr-spec> -i <if-addr> -s -l <dumplen> -t <addr-spec>
       -D <duration>
```

-a: transport address to receive data on; uses the following format
- <IPv4 address>/port
- /port
- <IPv6 address>/port
- <hostname>/port

May be specified repeatedly.

-i: address of the local interface to use for listening to multicast packets

-s: packet reports in short form: one line per packet:

reception timestamp (μs!), sender, receiver address, packet size

If “-s” is not specified, the long form is implied. In this case, the above line is followed by a hexdump of (parts of) each packet received:

```
000000  xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx    ................
```

-l: Number of bytes to include in the hexdump

-t: transport address to accept TCP connections to dump packets on

-D: duration to run (in seconds)

## Hints (1)

- Transport address(es) to receive data on
  - socket (SOCK_DGRAM, AF_INET, …)
  - Address type (unicast or multicast) can easily be determined from the class
    - For multicast, bind() to the proper port number with INADDR_ANY
    - Then, use setsockopt () with IP_ADD_MEMBERSHIP for the multicast and i/f address
    - If no i/f address was specified, just use INADDR_ANY
  - You MUST create and bind an individual UDP socket for every address
    - Otherwise, you cannot determine the destination of the packet later on
  - You need to parse the address arguments “by hand”
  - Remember host vs. network byte order

- Single line format, e.g.:

```
14:09:00.123456 134.102.218.59:40000 -> 134.102.218.58:47000 79 Bytes
```

  - **Remember again network vs. host byte order**

## Hints (2)

- Accepting a TCP connection to dump to
  - Use another listening socket socket (SOCK_STREAM, AF_INET, …)
  - accept () incoming connections
    - If there is more than one, distribute the data to all of them
  - A poll()/select() READ event and a subsequent read() result of “0” or “-1” indicate that the peer is gone.
    - Try this out, this varies between different operating systems
    - Close the socket locally
    - If none are left, revert to dumping to the screen (or no dumping at all)
  - For this assignment, use all sockets in blocking mode (default setting)
    - Using non-blocking i/o will make things far too complicated and cause extra headache
    - But would, nevertheless, be the right thing to do in practice
  - Test with telnet(1)
    - telnet 127.0.0.1 50000 if your process listens on port #50000

## Hints (3)

- Time 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 (4)

- Signals
  - 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
- Short note on hexdumps()
  - printf (“%x02x”, variable) prints the contents as 2 hex digits with leading zero
  - Exception: if the the highest bit is “1”, then leading “ffffff” may appear.
  - Solution: use (variable & 0x0ff) for printing (you care about 8 bits only)

## Hints (5)

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

if ((s = socket (AF_INET, SOCK_STREAM, 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);
}
i_addrlen = sizeof (i_addr);

if (listen (s, 3) == -1) {
   perror ("listen failed");
   exit (-1);
}
```

## Hints (6)

```
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 */
              }
}}}
```