308-435B TCP/IP programming

Juan de Lara, Hans Vangheluwe

McGill University

Winter Term 2001

Resources

1. Request For Comments (RFC):
   - structured overview TCP/IP related RFCs: http://www.webzone.net/machadospage/RFC.html
   - complete overview (with search): http://www.faqs.org/rfcs/
   - example RFCs:
     - the Transport Control Protocol (TCP) RFC 793: rfc793.txt
     - the User Datagram Protocol (UDP) RFC 768: rfc768.txt
2. man pages:
   - the Single UNIX specification (searchable): http://www.opengroup.org/onlinepubs/7908799/index.html
   - man on local machine
3. books:
   - UNIX network programming, Volume 1: Networking APIs: Sockets and XTI Richard Stevens, Prentice Hall, 1998, 2nd Edition.

   This text is mostly based on the above book and its earlier edition.

Network Layers

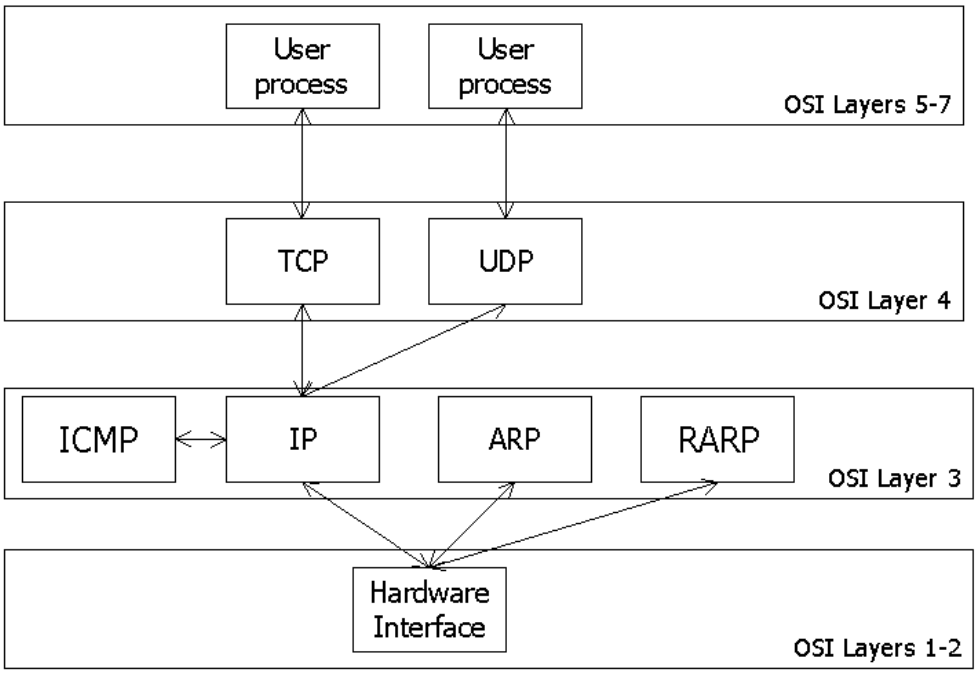
- Divide-and-conquer: network layers.
- 7 layer OSI (Open Systems Interconnection) model (later than TCP/IP):
  1. Physical: transmission of bit streams over physical medium.
  2. Data Link: adds reliability services to physical layer.
  3. Network: source-to-destination delivery across multiple networks.
  4. Transport: source-to-destination delivery of entire message.
  5. Session: network dialog controller.
  6. Presentation: syntax and semantics of exchangedinformation.
  7. Application: user access to the network.

Protocols at each level.

- TCP/IP model:
  1. Physical.
  2. Data Link.
  3. Network:
     - Internetworking Protocol (IP)
     - ICMP (Internet Control Message Protocol)
     - IGMP (Internet Group Message Protocol)
     - ARP (Address Resolution Protocol)
     - RARP (Reverse Address Resolution Protocol)
  4. Transport:
     - Transmission Control Protocol (TCP)
     - User Datagram Protocol (UDP)

     Communication (API) between Application and Transport layer: **sockets**.
  5. Application: OSI's Session, Presentation, Application.
     - FTP (File Transport Protocol) (TCP)
     - TFTP (Trivial File Transfer Protocol) (UDP)
     - TELNET (remote login)
     - SMTP (Simple Mail Transfer Protocol)
     - . . .

     Common architecture: **client-server**.



TCP/IP protocol suite

- 1980s: standard for the ARPANET and associated DoD networks
- 1987: protocol for communication in the NSFNET (supercomputers)

- now: protocol the ”Internet”.

Characteristics of the *protocol suite*:

- not vendor-specific
- implemented on almost all platforms
- used for both local and wide area networks
- wide use thanks to inclusion in the BSD Unix system around 1982.

User interaction with the TCP/IP protocol suite:

- IP is not accessed directly.
    - unreliable
    - connectionless
    - no error checking or tracking
    - data transported in packets called**datagrams**, transported separately $\rightarrow$
        - may travel along different routes
        - can arrive out of sequence
        - may be duplicated
- UDP/IP:
    - unreliable
    - connectionless
    - IP +
        - port numbers
        - length
        - optional checksum for verification
- TCP/IP:
    - reliable (checksums, positive acknowledgements, timeouts)
    - connection-oriented (establish - transmission - terminate)
    - full-duplex (end-to-end flow control)
    - byte-stream service (sequencing)

Internet addesses

- **uniquely** identify networks and computers
- addressing is protocol-specific
- IP (internet address): 32 bits, encoding network ID and host ID.

Related to but **not** the same as symbolic “domain” names such as www.cs.mcgill.ca.
Number of recipients:

- unicast address: single recipient
- multicast address: group of recipients
- broadcast address: all the host in the network (255.255.255.255)

Format classes (later):
Usually written as 4 dot-separated decimal numbers (*e.g.*, 132.206.51.10).

At tranport layer: add a **port number** (a 16-bit integer)
→ identify communicating **processes** in host.

TCP and UDP define well-known addresses (port numbers) for well-known *services*. In /etc/services:

```
daytime         13/tcp
daytime         13/udp
netstat         15/tcp
qotd            17/tcp          quote
msp             18/tcp                          # message send protocol
msp             18/udp                          # message send protocol
chargen         19/tcp          ttytst source
chargen         19/udp          ttytst source
ftp-data        20/tcp
ftp             21/tcp
fsp             21/udp          fspd
ssh             22/tcp                          # SSH Remote Login Protocol
ssh             22/udp                          # SSH Remote Login Protocol
telnet          23/tcp
# 24 - private
smtp            25/tcp          mail
```

Communicate with sendmail process on smtp (25) port on localhost:

```
telnet localhost smtp
```

Check active internet connections:

```
hv@lookfar 59% netstat --inet -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 HSE-Montreal-ppp34:1023 mimi.CS.McGill.CA:ssh   ESTABLISHED
tcp        0      0 *:X                     *:*                     LISTEN
tcp        0      0 *:www                   *:*                     LISTEN
tcp        0      0 *:https                 *:*                     LISTEN
tcp        0      0 *:587                   *:*                     LISTEN
tcp        0      0 *:smtp                  *:*                     LISTEN
tcp        0      0 *:printer               *:*                     LISTEN
tcp        0      0 *:ssh                   *:*                     LISTEN
tcp        0      0 *:finger                *:*                     LISTEN
raw        0      0 *:icmp                  *:*                     7
raw        0      0 *:tcp                   *:*                     7
```

socket addresses

- socket = API to network services
- UNIX: I/O by read/write from/to a file descriptor.
- file descriptor = integer associated with an open file
- open file can be a network connection, a FIFO, a pipe, a terminal, , *etc*.
- invoke socket() to get a socket
- types of sockets: DARPA Internet addresses (Internet Sockets), path names on a local node (Unix Sockets), CCITT X.25 addresses, , *etc*.

The socket address structure (in `<sys/socket.h>`)

```
struct sockaddr
{
    u_short sa_family;    /* address family: AF_XXX value */
    char    sa_data[14];  /* up to 14 bytes of protocol-specific address */
};
```

- `sa_family`: address family (`AF_INET`)
- `sa_data`: interpretation depends on the address family. In case of the Internet family: destination address and socket port number.

Easy to fill in using (in `<netinet/in.h>`):

```
struct in_addr
{
    u_long s_addr;              /* 32-bit address, in network byte order */
};

struct sockaddr_in
{
    short          sin_family;   /* AF_INET */
    u_short        sin_port;     /* 16-bit port number, in network byte order */
    struct in_addr sin_addr;     /* 32-bit address, in network byte order */
    char           sin_zero[8];  /* unused */
};
```

`u_short` and `u_long` are defined in `<sys/types.h>`.
For some API calls, an explicit cast from `struct sockaddr_in *` to `(struct sockaddr *)` in needed.
`sin_zero` (padding the structure to the length of `struct sockaddr`) must be set to all zeros (*e.g.*, with `memset()`).

Network and Host byte orders

Difference in storage order of integers' bytes on different machine architectures.
For example a 16-bit integer, made up of 2 bytes can be stored in two different ways:

- Little (low) endian: stores the low-order byte at the starting address.
- Big (high) endian: the high-order byte is stored at the staring address.

Note: this does not apply to character strings.

For networking: **network byte order**.

Conversion routines:

```
#include <sys/types.h>
#include <netinet/in.h>

u_long  htonl (u_long  hostlong);
u_short htons (u_short hostshort);
u_long  ntohl (u_long  netlong);
u_short ntohs (u_short netshoert);
```

- h stands for host
- n stands for network
- l stands for long
- s stands for short

`sin_addr` and `sin_port` fields *must* be in Network Byte Order as they get encapsulated in the packet at the IP and UDP layers, respectively.
`sin_family` is only used by the kernel to determine what type of address the structure contains, so it must be in Host Byte Order. It is not sent over the network.

Address convertion routines

An Internet address is usually written in the dotted-decimal format (*e.g.*, , `10.12.110.57`).
Conversion between dotted-decimal format (a string) and a `in_addr` structure:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr(char * ptr);

char * inet_ntoa(struct in_addr inaddr);
```

`inet_addr()` converts a character string in dotted-decimal notation to a 32-bit Internet address (in Network Byte Order). It returns -1 on error. Beware ! -1 corresponds to the IP address 255.255.255.255, the broadcast address.

Example: convert the IP address ”10.12.110.57” and store it

```
ina.sin_addr.s_addr = inet_addr("10.12.110.57");
if ( ina.sin_addr.s_addr == -1 )   /* error */
{
 ...                           /* error handling */
}
```

Remarks:

- `inet_ntoa()` takes a `struct in_addr` as argument, *not a long*.
- `inet_ntoa()` returns a `char *` pointing to a **statically** stored char array inside `inet_ntoa()`. The string will be overwritten at each call:

```
char *a1, *a2;
.
.
a1 = inet_ntoa(ina1.sin_addr);  // assume this holds 192.168.4.14
a2 = inet_ntoa(ina2.sin_addr);  // assume this holds 10.12.110.57
printf("address 1: %s\n",a1);
printf("address 2: %s\n",a2);
```

will print

```
address 1: 10.12.110.57
address 2: 10.12.110.57
```

Elementary Socket System Calls: *socket()*

Invoke `socket()` to specify the type of communication protocol desired (TCP, UDP, *etc.*).

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int family, int type, int protocol);
```

- `family` is set to `AF_INET`
- `type` is `SOCK_STREAM` for TCP and `SOCK_DGRAM` for UDP.

`socket()` returns

- a socket descriptor that can be used in later system calls,
- or -1 on error. Global variable `errno` is set to the error's value (use `perror()` to print msg).

TCP client/server architecture

Typical sequence of system calls to implement TCP clients and servers.

```
Server                                          Client

socket()
  |
  V
bind()
  |
```

```
    V
  listen()
    |
    V
  accept()                                        socket()
    |                                               |
blocks until connection from client                 V
    | <-- connection establishment ------------> connect()
    |                                               |
    V                                               V
  read() <-------- data (request)--------------- write()
    |                                               |
process request                                     |
    |                                               |
    V                                               V
  write() -------- data (reply) ----------------> read()
```

The *bind()* system call

`bind()` assigns a name to an unnamed socket.
Associates the socket with a *port* in the local machine.
The port number is used by the kernel to match an incoming packet to a certain process's socket descriptor.
Used by TCP and UDP servers and by UDP clients.

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

`sockfd` is the socket file descriptor returned by `socket()`.
`my_addr` is a pointer to a `struct sockaddr`.

- port
- IP address

May need to **cast** pointer to `struct sockaddr *`
`addrlen` can be set to `sizeof(struct sockaddr)`.
Returns -1 on error and sets `errno` to the error's value.
Code fragment be necessay to set up a TCP server (will be completed later):

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define MYPORT 3490
```

```
int main()
{
  int                sockfd;
  struct sockaddr_in my_addr;

  if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
  {
    perror("socket");
    exit(1);
  }

  my_addr.sin_family      = AF_INET;         /* host byte order */
  my_addr.sin_port        = htons(MYPORT);  /* short, network byte order */
  my_addr.sin_addr.s_addr = htonl(INADDR_ANY);  /* my own IP address */
  memset(&(my_addr.sin_zero), '\0', 8);  /* zero the rest of the struct */

  if ( bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) < 0 )
  {
    perror("bind");
    exit(1);
  }
    .
    .
    .
```

Automating getting the local IP address and/or port:

```
  my_addr.sin_port = 0;           /* choose an unused port at random */
  my_addr.sin_addr.s_addr = htonl(INADDR_ANY);  /* use my IP address */
```

Ports below 1024 are reserved. Upto 65535 (`sin_port` is 16 bits long) can be used, provided not in use by another process.

When trying to rerun a server, `bind()` often fails

```
"Address already in use."
```

Probably a socket that was connected has not been closed properly, is still “alive” in the kernel and is occupying the port. Two options are possible:

- wait for it to clear (a minute or so), or
- add code to the program allowing it to reuse the port.

```
int yes=1;

if (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1)
{
  perror("setsockopt");
  exit(1);
}
```

The *connect()* system call

- TCP client
- connect a socket descriptor (after `socket()`)
- establishes connection with a server

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

- `connect()` returns -1 on error and sets `errno`
- `sockfd` is a socket file descriptor returned by `socket()`
- `serv_addr` points to a structure with *destination* port and IP address
- `addrlen` set to `sizeof(struct sockaddr)`

Initial code necessay for a TCP client:

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define DEST_IP   "150.244.56.39"
#define DEST_PORT 13

main()
{
  int sockfd;
  struct sockaddr_in dest_addr;   /* will hold the destination addr */

  if ((sockfd = socket(AF_INET, SOCK_STREAM, 0))<0)
  {
    perror("socket");
    exit(1);
  }

  dest_addr.sin_family      = AF_INET;            /* inet address family */
  dest_addr.sin_port        = htons(DEST_PORT);   /* destination port */
  dest_addr.sin_addr.s_addr = inet_addr(DEST_IP); /* destination IP address */
  memset(&(dest_addr.sin_zero), '\0', 8);   /* zero the rest of the struct */

  if (connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr))<0)
  {
    perror("connect");
    exit(1);
  }
    .
```

```
.
.
```

The code does **not** call `bind()`.

We don't care about our local port number, only about the remote port.

The kernel will choose a local port, and the site we connect to will *automatically* get this information from us.

A connectionless client (UDP) can also use `connect()`. In this case, the system call just stores the `serv_addr` specified by the process, so that the system knows where to send any future data the process writes to the `sockfd` descriptor. Also, only datagrams from this address will be received by the socket.

not take place. The advantage of connecting a socket if we use UDP, is that the destination address is not needed for every datagram sent.

The *listen()* system call

- used by a TCP server
- specify how many client connections can be waiting while the server is servicing other clients.
- incoming connections wait in this queue until `accept()` is invoked.

```
int listen(int sockfd, int backlog);
```

Returns -1 and sets `errno` on error.

Example:

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define MYPORT 3490

int main()
{
  int                sockfd;
  struct sockaddr_in my_addr;

  if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
  {
    perror("socket");
    exit(1);
  }

  my_addr.sin_family      = AF_INET;
  my_addr.sin_port        = htons(MYPORT);
  my_addr.sin_addr.s_addr = inet_addr(INADDR_ANY);
  memset(&(my_addr.sin_zero), '\0', 8);   /* zero the rest of the struct */

  if ( bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) < 0 )
```

```
  {
    perror("bind");
    exit(1);
  }

  if ( listen(sockfd, 5) < 0 )
  {
    perror("listen");
    exit(1);
  }
    .
    .
    .
```

The *accept()* system call

- TCP servers
- after calling `listen()`
- gets queued connection
- returns a *new* socket file descriptor
- use for this connection (send/receive data)
- still listening on original socket

```
#include <sys/socket.h>

int accept(int sockfd, void *addr, int *addrlen);
```

- `sockfd` is the socket descriptor
- `addr` points to a local `struct sockaddr_in`
- struct will be filled with the information about the incoming client
- `*addrlen` should be set to `sizeof(struct sockaddr_in)`
- `accept` will not put more than that many bytes into `*addr`
- may put less, will then modify `addrlen`
- returns -1 and sets `errno` if an error occurs.

Continuation of the example:

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define MYPORT 3490

int main()
{
  int                 sockfd, csd, len;
  struct sockaddr_in my_addr, cliaddr;
```

```
if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
   perror("socket");
   exit(1);
}

my_addr.sin_family      = AF_INET;
my_addr.sin_port        = htons(MYPORT);
my_addr.sin_addr.s_addr = inet_addr(INADDR_ANY);
memset(&(my_addr.sin_zero), '\0', 8);   /* zero the rest of the struct */

if ( bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) < 0 )
{
   perror("bind");
   exit(1);
}

if ( listen(sockfd, 5) < 0 )
{
   perror("listen");
   exit(1);
}

len = sizeof(cliaddr);
while(1)
{
  if ((csd = accept ( sockfd, (struct sockaddr *)&cliaddr, &len))<0)
  {
     perror("accept()");
     exit(1);
  }
  printf("connection from %s, port %d\n",
          inet_ntoa(cliaddr.sin_addr),
          ntohs(cliaddr.sin_port));
  .
  .
  .
```

send() and *recv()* system calls

Similar to `write()` and `read()` but give more control.

Used for sending and receiving over TCP or connected datagram sockets.

```
#include <sys/types.h>
#include <sys/socket.h>

int send(int sockfd, const void *msg, int len, int flags);
```

```
int recv(int sockfd, const void *msg, int len, int flags);
```

- `sockfd` is the socket descriptor to send or receiv data
- `msg` is pointer to buffer for send/receive data
- `len` is the length of that data in bytes, or the maximim size of the receiving buffer
- `flags` usally set to zero

```
MSG_OOB        /* send or receive out-of-band data */
MSG_PEEK       /* peek at incoming message (recv or recvfrom) */
MSG_DONTROUTE  /* bypass routing (send or sendto) */
```

Both system calls return the *length* of the data that was sent or received, or -1 on error.

If `recv()` returns 0,that means that the server has closed the connection.

close() and *shutdown* system calls

The usual Unix `close()` is also used to close a socket. The prototype is the following:

```
int close(int fd);
```

This will prevent any more reads and writes to the socket. Some process attempting to read or write the socket on the remote end will receive an error.

`shutdown()` allows more control over how the socket closing. It permits to cut off communication in a certain direction, or both ways (like `close()`). The prototype is:

```
int shutdown(int sockfd, int how);
```

`sockfd` is the socket file descriptor to be closed. `how` is one of the following:

- 0 – Further receives are disallowed
- 1 – Further sends are disallowed
- 2 – Further sends and receives are disallowed (like *close()*)

`shutdown()` returns 0 on success, and -1 on error (with `errno` set accordingly)

The complete example

The following is the code for the daytime client:

```
#include <stdio.h>         /* for printf() */
#include <string.h>        /* for memset() */
#include <stdlib.h>        /* for exit()   */
#include <unistd.h>        /* for close()  */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
/* get the time in Spain */
/*#define DEST_IP  "150.244.56.39"*/
/*#define DEST_PORT 13            */

/* get the time locally */
#define DEST_PORT 3490

#define MAXLINE   128
#define MAX_QUERY 15

int
main(void)
{
  int sockfd;                   /* socket file descriptor to comm through */
  struct sockaddr_in dest_addr; /* the server's address */
  char recvline[MAXLINE + 1];   /* to store received data in */
  int num_recvd;                /* number of bytes received */

  int cnt;                      /* index variable for series of time queries */
  for (cnt=0; cnt<MAX_QUERY; cnt++)
  {
   if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   {
     perror("socket");
     exit (1);
   }

   /* set up the destination address sockaddr_in structure */

    /* Internet family */
    dest_addr.sin_family = AF_INET;         /* in HBO */

    /* to communicate with the server in Spain, destination IP address */
    /* dest_addr.sin_addr.s_addr = inet_addr(DEST_IP); long, in NBO */

    /* to communicate with our own server */
    /* server runs on the local host, IP address is filled automatically */
    dest_addr.sin_addr.s_addr = htonl(INADDR_ANY);  /* long, in NBO */

    /* at the above IP address, port DEST_PORT */
    dest_addr.sin_port = htons(DEST_PORT);  /* short, in NBO */

    /* zero the rest of the struct */
    memset(&(dest_addr.sin_zero), '\0', 8);

   /* try to set up a connection */
   if (connect(sockfd, (struct sockaddr *) &dest_addr,
       sizeof(struct sockaddr)) < 0)
```

```
   {
     perror("connect");
     exit (1);
   }

   /* connection blocks until data is received */
   if ( (num_recvd = recv(sockfd, recvline, MAXLINE, 0)) < 0)
   {
     perror("recv");
     exit (1);
   }

   recvline[num_recvd] = '\0';         /* turn it into a string */
   if (fputs(recvline, stdout) == EOF)
   {
     perror("fputs error");
     exit (1);
   }

   /* flush the output stream to print right away */
   fflush(stdout);

   /* close the socket */
   close(sockfd);
  }

  return (0);

}
```

The following is the code for the daytime server:

```
#include <stdio.h>         /* for printf() */
#include <string.h>        /* for memset() */
#include <stdlib.h>        /* for exit()   */
#include <unistd.h>        /* for close()  */
#include <time.h>          /* for time()   */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "snprintf.h"

#define MYPORT  3490
#define MAX_LINE 128
#define MAX_QUEUE 5
#define MAX_CONNECTS 20
```

```
int
main(void)
{
  int sockfd;          /* socket (file) descriptor */
  int acsd;            /* accepted connection socket descriptor */
  struct sockaddr_in my_addr;     /* this server's address */
  struct sockaddr_in client_addr; /* a client's address */
  socklen_t len=sizeof(client_addr); /* length of the client_addr structure */
  int yes=1;           /* for socket options */
  int in_conn;         /* counter for incoming connections */
  time_t ticks;        /* to calculate current time and date */
  char buff[MAX_LINE]; /* to store the server's answer before sending */

  /* try to create a socket */
  if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
  {
    perror("socket");
    return -1;
  }

  /* set up the my_addr sockaddr_in structure */

   /* Internet family */
   my_addr.sin_family = AF_INET;     /* in HBO */

   /* will listen on this host's IP address */
   my_addr.sin_addr.s_addr = htonl(INADDR_ANY); /* long, in NBO */

   /* will listen on MYPORT port */
   my_addr.sin_port = htons(MYPORT); /* short, in NBO */

   /* zero the rest of the struct */
   memset(&(my_addr.sin_zero), '\0', 8);

  /* make the socket (port MYPORT) re-usable for bind()            */
  /* otherwise, the kernel may still hang on to the port           */
  /* and bind() will fail. Without SO_REUSEADDR, just have to wait */
  if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int))==-1)
  {
    perror("setsockopt");
    exit(1);
  }

  /* associate the socket with a port (MYPORT) on this machine */
  if ( bind (sockfd, (struct sockaddr *) &my_addr,
       sizeof(struct sockaddr)) < 0)
  {
    perror("bind");
```

```
  exit(1);
}

/* specify how many incoming connection client connections
 * will be queued
 */
if (listen(sockfd, MAX_QUEUE) < 0)
{
  perror("listen");
  exit(1);
}

/* accept MAX_CONNECTS incoming connections to this server */
for (in_conn=0; in_conn<MAX_CONNECTS; in_conn++)
{
  /* accept an incoming connection,
   *  get a socket descriptor: acsd
   *  get information about the client in client_addr
   */
  if ( (acsd = accept(sockfd, (struct sockaddr *) &client_addr, &len)) < 0)
  {
    perror("accept()");
    exit(1);
  }
  printf("connection from client %s, port %d\n",
         inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));

  /* sleep a while */
  sleep(20);

  /* prepare the reply */

  /* obtain current time
   * in seconds since the Epoch (00:00:00 UTC, January 1, 1970)
   */
  ticks = time(NULL);

  /* convert the time to ASCII and put in a string buffer */
  snprintf(buff, sizeof(buff), "%.24s\n", ctime(&ticks));

  /* send the reply string to the client, null terminated, flags=0 */
  send(acsd, (void *) buff, strlen(buff) + 1, 0);

  /* close the socket through which the client is accessed */
  close(acsd);
 }

/* close the server's socket
```

```
   * attempts to read/write the socket on the client side will
   * receive an error.
   */
  close(sockfd);

  return (0);
}
```

Concurrent servers

After a connection from a client has arrived (when the `accept()` system call returns control) a server has two choices:

- For an *iterative server*:
  Process the request and send the reply.
  Used when the clients' requests can be handled *quickly* and in a *single* response.
  This is the type of servers we used upto now.
- For a *concurrent server*:
  A new process is created to service the client (typically using the `fork()` system call).
  This new process handles *only* the incoming client request and does not have to respond *other* client requests.
  When the dedicated (to the client) process finishes, it closes its communication channel with the client and terminates.
  The parent of this process returns to wait for new client requests in the `accept()` system call.

Typical pseudocode for concurrent servers is:

```
  sock = socket(...);

  bind(sock, ...);

  listen(sock, ...);

  while(1)
  {
    new_sock = accept (sock, ...);
    switch (fork())    /* Create new process to handle client request */
    {
      case -1:                          /* error */
                                        /* perform some error actions */
        break;
      case 0:                           /* child */
        close (sock);                   /* close listening socket */
        process_request(new_sock,...);/* process the request */
        close (new_sock);               /* close connected socket */
        exit (0);
      default:                          /* parent, return to accept */
```

```
        close (new_sock);             /* close connected socket */
        break;
    }
  }
```

It is common to catch the SIGCHLD signal in order to perform some clean up actions whenever a child finishes and above all to avoid *zombie* processes.

Example: modify the previous daytime server to make it concurrent.

In order to be able to connect several clients at a time, we put the serving processes to sleep during a few seconds.

The client code is equal to that of the client studied in the past sections.

The following is the code for the concurrent daytime server:

```
#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <stdlib.h>        /* for exit()   */
#include <unistd.h>        /* for close()  */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>         /* for errno */
#include "snprintf.h"

#define MYPORT  3490
#define MAXLINE 128
#define TSLEEP  3

/*
 * signal handler
 *
 *  SIGCHLD: child process (servicing a client) stopped or terminated,
 *           must be handled to avoid zombies
 *           default action: ignored
 *
 *  SIGINT: interrupt from keyboard
 *          default action: terminate process
 *          We silently assume INT signals are sent
 *          to the parent process only.
 */
void
signal_handler(int signum)
```

```
{
  int status;

  if (signum == SIGINT)
  {
   fprintf(stderr, "Interrupt from keyboard, server terminating\n");
   fflush(stderr);
   exit(0);
  }

  if (signum == SIGCHLD)
  {
   pid_t child_pid; /* the ending child's PID */

   /* while processing, ignore further SIGCHLD interrupts */
   signal(SIGCHLD, SIG_IGN);

   child_pid = wait(&status);
   /*
    The wait function suspends execution of the current process
    until a child has exited, or until a signal is delivered  whose
    action is to terminate the current process or to call a  signal
    handling  function. If a child has already exited by the
    time of the call (a so-called "zombie" process), the function
    returns immediately.  Any resources used by the child are freed.
    */
   if (WIFEXITED(status))
   {
    fprintf(stdout, "\nfork()ed server child %d exited normally\n", child_pid);
    fflush(stdout);
   }
   else
   {
    fprintf(stderr, "Hmm ... child did not exit normally\n");
    fflush(stderr);
   }

   printf("Client serviced, process terminated\n\n");
   fflush(stdout);

   signal(SIGCHLD, signal_handler); /* capture this signal again */
  }

  return;
}

/*
 * process_request: processes a client's request:
```

```
 *                  send the date and time.
 */
void
process_request(int sock, const struct sockaddr_in *cliaddr)
{
  time_t ticks;         /* Used to calculate current time and date */
  char buff[MAXLINE];   /* Used to store server answer */

  printf("connection from %s, port %d\n",
         inet_ntoa(cliaddr->sin_addr), ntohs(cliaddr->sin_port));

  printf("Going to sleep (%d secs)\n", TSLEEP);

  sleep(TSLEEP);

  /* obtain current time */
  ticks = time(NULL);

  /* put the time in a string */
  snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));

  /* send the string, '\0' terminated */
  send(sock, (void *) buff, strlen(buff) + 1, 0);

  printf("Sending reply to client now\n");
}

int
main(int argc, char *argv[])
{
  int sockfd; /* server's listening socket */
  int csd;    /* client-connection sockets */
  struct sockaddr_in my_addr; /* this server's address */
  struct sockaddr_in cliaddr; /* the client's address (returned by accept) */
  socklen_t len = sizeof(cliaddr);
  int yes=1;
  pid_t pid;  /* process ID returned by fork() */

  /* catch keyboard interrupt signal to halt the server */
  signal(SIGINT, signal_handler);

  /* catch child normal termination (avoid zombies) */
  signal(SIGCHLD, signal_handler);

  if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
  {
    perror("socket");
    return -1;
```

```
}

my_addr.sin_family = AF_INET; /* host byte order */
my_addr.sin_port = htons(MYPORT); /* short, network byte order */
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
memset(&(my_addr.sin_zero), '\0', 8); /* zero the rest of the struct */

if (bind (sockfd, (struct sockaddr *) &my_addr, sizeof(struct sockaddr)) < 0)
{
  perror("bind");
  return -1;
}

/* make the socket (port MYPORT) re-usable for bind()            */
/* otherwise, the kernel may still hang on to the port           */
/* and bind() will fail. Without SO_REUSEADDR, just have to wait
 */
if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int))==-1)
{
    perror("setsockopt");
    exit(1);
}

if (listen(sockfd, 5) < 0)
{
  perror("listen");
  exit (1);
}

while (1) /* forever */
{
  /* blocks until an incoming connection request */
  if ((csd = accept(sockfd, (struct sockaddr *) &cliaddr, &len)) < 0)
  {
   /* The accept() system call is "slow" and may be interrupted
    * by a signal from a terminating child !
    * Not all kernels automatically restart interrupted system
    * calls. We do it here explicitly.
    */
   if (errno == EINTR)
    continue;
   else
   {
    perror("accept()");
    return -1;
   }
  }
```

```
pid = fork(); /* spawn a new process to handle client requests */

if (pid <0) /* error in creating a child process */
{
  perror("fork()");
  exit(1);
};

if (pid == 0) /* this is the child */
{
  /* Mark the listening socket as closed and return immediately.
   * The socket is no longer usable by this process.
   * The child does not need to access the listening socket,
   * only the parent does. The parent does not close this socket.
   */
  close(sockfd);

  /* process the request */
  process_request(csd, &cliaddr);

  /* As the request has been processed, this process is finished.
   * Mark the connected socket as closed and return immediately.
   * The socket is no longer usable by this process.
   *
   * Closing csd is not absolutely necessary as exit(0) ends the
   * process. The kernel will then close all open file descriptors.
   *
   */
  close(csd);

  /* the child terminates */
  exit(0);
};

if (pid > 0) /* this is the parent */
{
  /* Mark the connected socket as closed and return immediately.
   * The parent process only listens for new connections and
   * hence does not need the client-connection socket.
   * The latter is handled by the client.
   *
   * Closing csd by the parent does NOT terminate the
   * connection (of the child) with the client.
   * A FIN is only sent to the client when the socket reference
   * count reaches 0.
   */
  close(csd);
}
```

```
    /* of the above three cases, only (pid >0) parent continues
     * the infinite loop.
     */
  }
}
```

I/O Multiplexing and *select()*

To handle *multiple* sockets, different alternative solutions:

1. Make all the sockets non-blocking.
   Execute a *polling* loop reading the state of each socket.
   → waste of compute time.

2. `fork()` one child process to handle each I/O channel.

3. Asynchronous I/O using signals.
   May catch the SIGIO signal. Does not tell us which descriptor is ready for I/O.

4. Use the `select()` system call.
   Makes it possible to wait for any of *multiple events* to occur
   wake up the process only if one of these events occurs.

The function prototype for `select()` and related functions:

```
#include <sys/types.h>
#include <sys/time.h>

int select(int maxfdpl,
           fd_set * readfds, fd_set * writefds, fd_set * exceptfds,
           struct timeval * timeout);

FD_ZERO (fd_set * fdset);            /* clear all bits in fdset */
FD_SET  (int fd, fd_set * fd_set);   /* turn the bit for fd on in fdset */
FD_CLR  (int fd, fd_set * fd_set);   /* turn the bit for fd off in fdset */
FD_ISSET(int fd, fd_set * fd_set);   /* test the bit for fd in fdset */
```

The structure pointed by the `timeout` argument is defined in `<sys/time.h>` in the following way:

```
struct timeval
{
   long tv_sec;   /* seconds */
   long tv_usec;  /* microseconds */
};
```

Call `select()` to know when

- any of the descriptors in `readfds` is ready for reading,

- any of the descriptors in `writefds` is ready for writing,
- or any descriptor in `exceptfds` has any exceptional condition.

If not interested in one of these sets, use `NULL`.
`select()` can be set to return:

- Immediatelly after checking the descriptors, like a poll.
  For this the structrue pointed by `timeout` must be set to 0 seconds.
- When one of the descriptors is ready for I/O, but not waiting more than a certain time, specified by `timeout`.
- When one of the descriptors is ready for I/O, without time restrictions.
  For this, the `timeout` argument must be set to `NULL`.

The *maxfdpl* argument must be set to the value of the largest file descriptor in the sets *plus one*.

Initially, all the file descriptors in the sets must be set (with `FD_SET`).

When the function returns, if the descripor is set, then it is ready for I/O.

As usual, the function returns a value of -1 indicating an error.

The following code fragment declares a descriptor set and adds descriptors 1 and 5. Then test these descriptor for reading up to 3 seconds.

```
.
.
.
struct timeval     tv;
fd_set             fdvar;

tv.tv_sec  = 3;
tv.tv_usec = 0;

FD_ZERO (&fdvar);          /* initialize the set - all bits off */
FD_SET  (1, &fdvar);       /* turn on bit for fd 1 */
FD_SET  (5, &fdvar);       /* turn on bit for fd 5 */

if ( select(6, &fdvar, NULL, NULL, & tv) < 0 )
{
   perror ("select");
   return -1;
}

if ( FD_ISSET(1, &fdvar) ) /* we have received something on socket 1*/
{
   ...
}
```

```
if ( FD_ISSET(5, &fdvar) ) /* we have received something on socket 5*/
{
   ...
}
.
.
.
```

UDP sockets

UDP provides a connectionless service: each datagram sent is independent one. There is no relationship between sent datagrams even if they come from the same application.
Datagrams are *not numbered* which implies they

- can be delivered out of sequence, and
- can travel along different routes.

UDP does not break application data to fit in different datagrams, each request from the application layer must be *small enough* to fit into one user datagram.

UDP is also *unreliable*, there is no flow control nor windowing mechanism. There is no error control except for the checksum (the sender does not know if a message has been lost or duplicated).

UDP is used for processes that require simple request/reponse communication, with little concern for flow and error control. It is also suitable for applications with internal flow and error control mechanisms such as TFTP, and for broadcasting and multicasting purposes. UDP is used in management processes such as SNMP and in some routing updating protocols such as RIP.

UDP datagrams

UDP packets are called user datagrams, and have a fixed size of eight bytes. The fields it contains are:

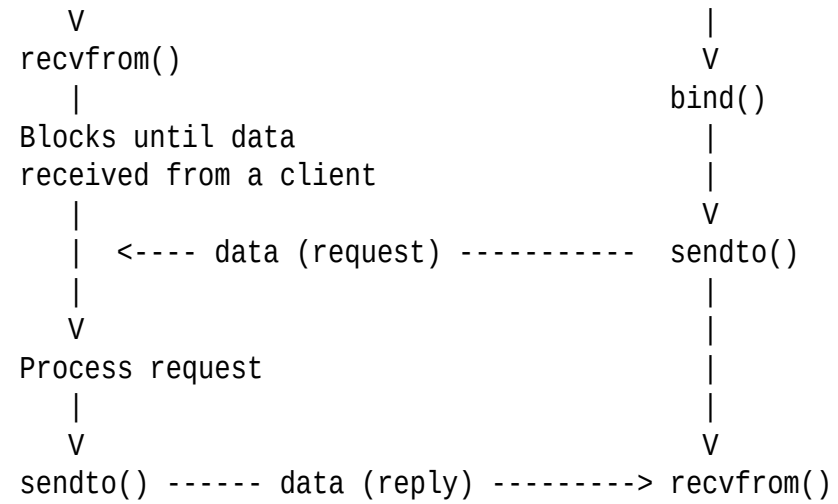
- Source port number.
- Destination port number.
- Length of the user datagram (16-bit).
- Checksum to detect errors over the entire user datagram.

The UDP client/server architecure

For a client-server using a connectionless protocol, the system calls are different from the connection oriented case. The next diagram shows a typical sequence of systems calls for both the client and the server.

```
Server                                  Client

socket()
   |
   V
 bind()
   |                                    socket()
```



The next section explains the `recvfrom()` and `sendto()` system calls, the other system calls have been studied in previous sections.

recvfrom() and *sendto()* functions

Since datagram sockets are not connected to a remote host, we need the destination address in both calls. Their prototypes are:

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
           const struct sockaddr *to, int tolen);

int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
             struct sockaddr *from, int *fromlen);
```

These calls are basically the same as the calls to `send()` and `recv()` with the addition of two other pieces of information—

- `to` is a pointer to a `struct sockaddr` which contains the destination IP address and port. `tolen` can be set to `sizeof(struct sockaddr)`. `sendto()` returns the number of bytes actually sent, or -1 on error.
- `from` is a pointer to a local `struct sockaddr` which will be filled with the IP address and port of the originating machine. `fromlen` is a pointer to a local `int` which should be initialized to `sizeof(struct sockaddr`. When `recvfrom()` returns, `fromlen` will contain the length of the address actually stored in `from`. `recvfrom()` returns the number of bytes received, or -1 on error (with `errno` set accordingly.)

Datagrams socket can also be connected (by calling `connect()`), and then `send()` and `recv()` can be used for all the communications. The socket itself is still a datagram socket and the packets still use UDP, but the socket interface will automatically add the destination and source information.

UDP example

An example of an UDP echo client and server is presented. The client code is as follows:

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>

#define MAXLINE 256
#define PORT    7000

int
main(int argc, char *argv[])
{
  int fsd; /* socket descriptor */
  int numsent; /* number of bytes sent */
  int numrec; /* number of bytes received from server */
  struct sockaddr_in servaddr; /* To be filled with server address */
  struct sockaddr_in replyaddr;
  socklen_t len= sizeof(servaddr); /* store address size */
  char sendline[MAXLINE];      /* string to be sent */
  char recvline[MAXLINE + 1];  /* received string */

  /* get a datagram (UDP) socket */
  if ((fsd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
  {
    perror("socket");
    exit(1);
  }

  /* fill address completely with 0's */
  memset(&servaddr, '\0', sizeof(servaddr));
  servaddr.sin_family = AF_INET; /* Internet family */
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY); /* this machine's IP address */
  servaddr.sin_port = htons(PORT); /* echo service port */
                                   /* standard: port 7 */

  while (1) /* forever */
  {
    /* prompt the user */
    printf("==>");
    fflush(stdout);

    /* read user input */
    fgets(sendline, MAXLINE, stdin);

    /* stop when "exit" is entered */
    if (strcmp(sendline, "exit\n") == 0)
```

```
     break;

    /* send entered text to the server (address in servaddr)
     * note how strlen(sendline) does not include the '\0'
     * at the end of the string
     */
    numsent = sendto(fsd, sendline, strlen(sendline), 0,
              (struct sockaddr *) &servaddr, sizeof(servaddr));
    if (numsent == -1)
    {
      /* note how only sender-side  errors are detected
       * UDP is connectionless, don't if care datagram arrives
       */
      perror("sendto");
      exit(1);
    }

    /* if we asked the server to die, don't wait for reply */
    if (strcmp(sendline, "die!\n") != 0)
    {
     /* Receive the answer from the server */
     if ((numrec = recvfrom(fsd, recvline, MAXLINE, 0, &replyaddr, &len)) < 0)
     {
       perror("recvfrom");
       exit(1);
     }

     /* the '\0' at the end of sendline was not sent
      * need to add one now to print as string
      */
     recvline[numrec] = '\0';
     printf(" from server: %s\n", recvline); /* show on screen */
    }
  }

  /* close the socket */
  close(fsd);

  /* normal end */
  printf("echo client normal end\n");

  return 0;
}
```

The server code is as follows:

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>

#define MAXLINE 256
#define PORT    7000

int
main(int argc, char *argv[])
{
  int fsd;        /* socket descriptor */
  int numrec;     /* number of bytes received */
  struct sockaddr_in servaddr; /* To be filled with server address */
  struct sockaddr_in cliaddr;  /* To be filled with client address */
  socklen_t len=sizeof(cliaddr); /* size of client address structure */
  char recvline[MAXLINE + 1];    /* area to receive in */

  /* get a datagram (UDP) socket descriptor */
  if ((fsd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
  {
    perror("socket");
    exit(1);
  }

  memset(&servaddr, '\0', sizeof(servaddr)); /* fill address with 0's */
  servaddr.sin_family = AF_INET; /* Internet family */
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY); /* My IP address */
  servaddr.sin_port = htons(PORT); /* port on which
                                    * this service is provided
                                    * wellknown: 7 */

  /* Assign an address to the socket */
  bind(fsd, (struct sockaddr *) &servaddr, sizeof(servaddr));

  /* forever */
  while (1)
  {
    /* block until an incoming client request */
    if ( (numrec = recvfrom(fsd, recvline, MAXLINE, 0,
                  (struct sockaddr *) &cliaddr, &len)) < 0)
    {
      perror("recvfrom");
      exit(1);
    }
```

```
    fprintf(stdout,
            "connection from %s, port %d. Received %d bytes.\n",
            inet_ntoa(cliaddr.sin_addr), /* IP address, as a string */
            ntohs(cliaddr.sin_port),     /* port number, in HBO */
            numrec);                     /* number of bytes received */
    fflush(stdout);

    recvline[numrec]='\0'; /* make it a string */
    if (strcmp(recvline, "die!\n") == 0)
    {
      fprintf(stdout, "a client asks me to terminate, complying\n");
      fflush(stdout);
      break;
    }

    /* echo the received to the client */
    if (sendto(fsd, recvline, numrec, 0, &cliaddr, len) < 0)
    {
      perror("sendto");
      exit(1);
    }
  }

  fprintf(stdout, "server normal end\n");
  fflush(stdout);

  return 0;
}
```

Makefile

The Makefile to produce all examples is here. The file snprintf.h needs to be included to avoid compilation warnings (at least on Linux 2.2.16-22).