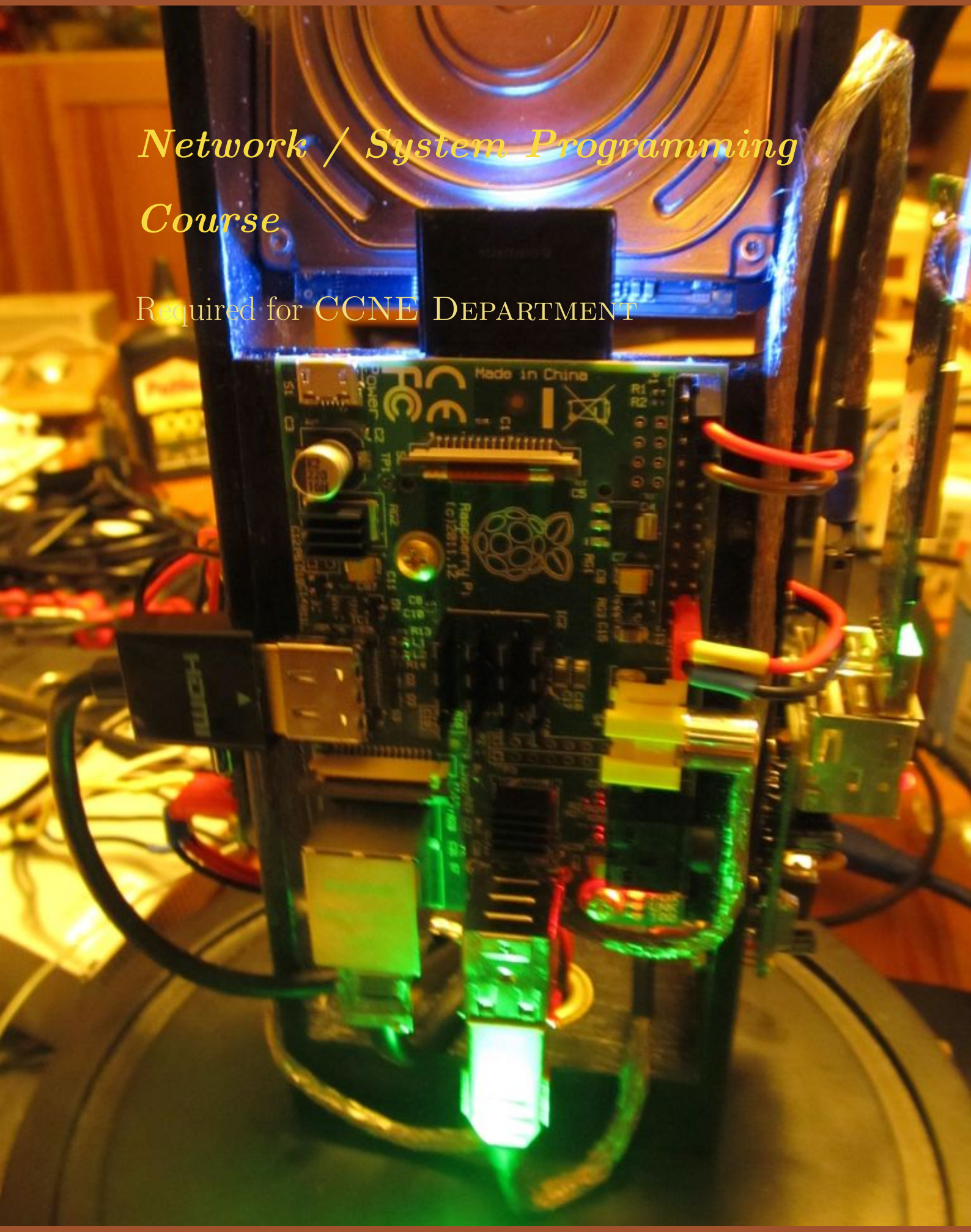
Lebanese University - UIT Saida




# *Network / System Programming Course*

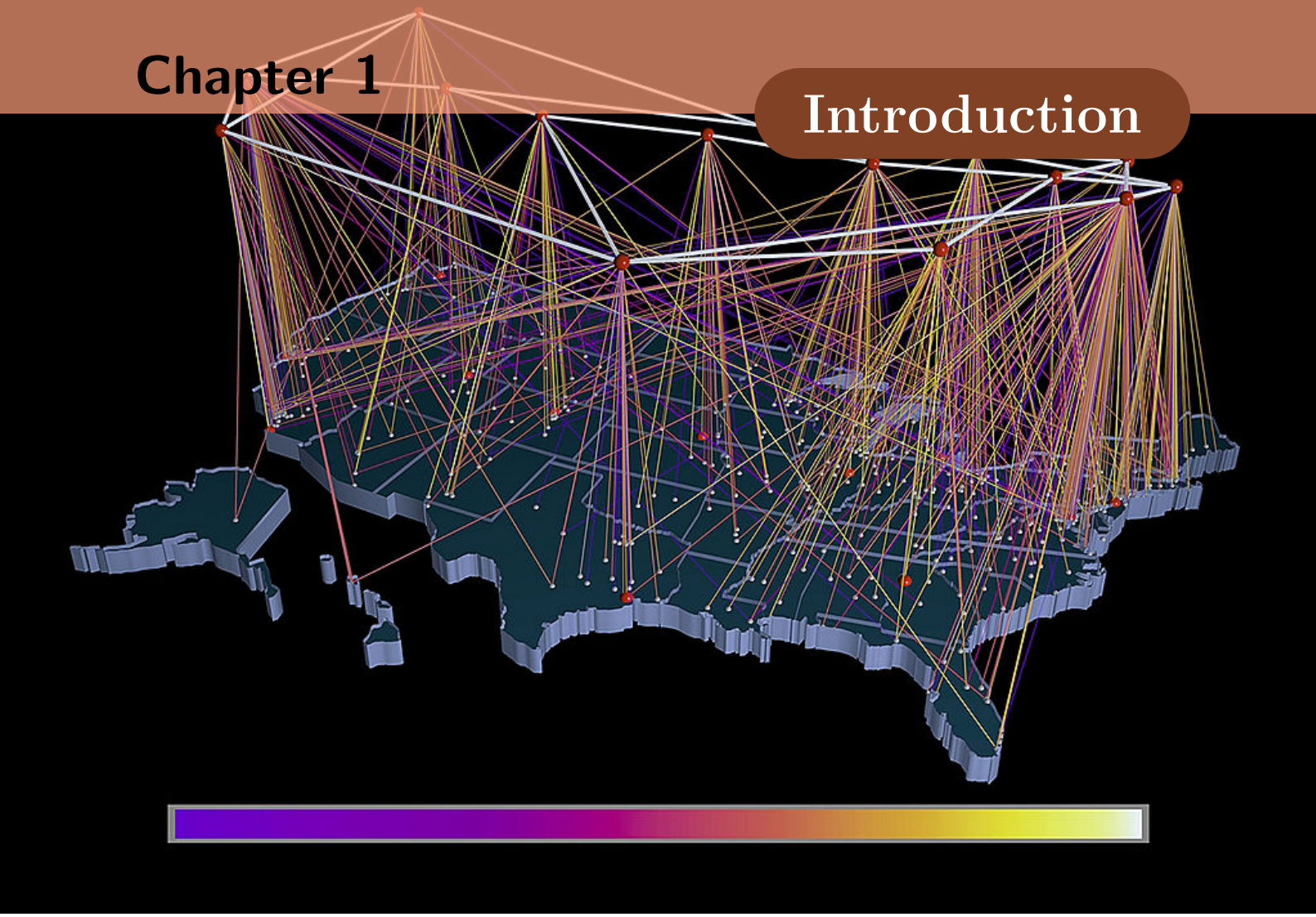
Required for CCNE Department

Dr. Abdelsalam Hajjar & Dr. Hussein EL Ghor

# Contents

# Chapter 1

# Introduction



## 1.1 Client/Server topology

The client/server model is a distributed application that partitions tasks between:

- The providers of a resource or service, called servers.

**Definition 1** (Server)**.**

A server machine is a host that is running one or more server programs which share their resources with clients.

- The service requesters, called clients.

**Definition 2** (Client)**.**

A client does not share any of its resources, but requests a server's service. Clients therefore initiate communication sessions with servers which await incoming requests.

Clients and servers often communicate over a computer network on separate hardware, but both clients and servers may reside in (exist in) the same system.


Figure 1.1: Schedule produced by EDeg

## 1.2 OSI Architecture

(OSI) Open System Interconnection is a communications architecture that defines standards for linking diverse computers. The OSI model consists of seven layers: Each layer performs a related subset of the functions required to communicate with another system.

The seven layers are as follows:

- **Layer 7: Application Layer**

  Defines interface to user processes for communication and data transfer in network.

- **Layer 6: Presentation Layer**

  Masks the differences of data formats between dissimilar systems.

  Encodes and decodes data; Encrypts and decrypts data; Compresses and decompresses data.

- **Layer 5: Session Layer**

  Manages user sessions and dialogues.

  Controls establishment and termination of logic links between users.

- **Layer 4: Transport Layer**

  Manages end-to-end message delivery in network.

- **Layer 3: Network Layer**

  Determines how data are transferred between network devices.

  Routes packets according to unique network device addresses.

  Ex: Routers.

- **Layer 2: Data Link Layer**

  Defines procedures for operating the communication links.

  Detects and corrects packets transmit errors.

  Ex: Switches.

- **Layer 1: Physical Layer**

  Defines physical means of sending data over network devices.

  Interfaces between network medium and devices.

  Ex: Hubs.

This example illustrates how a message is sent from one node of a network to another. As the message goes through the different OSI layers, headers and trailers are added and removed to and from the message.

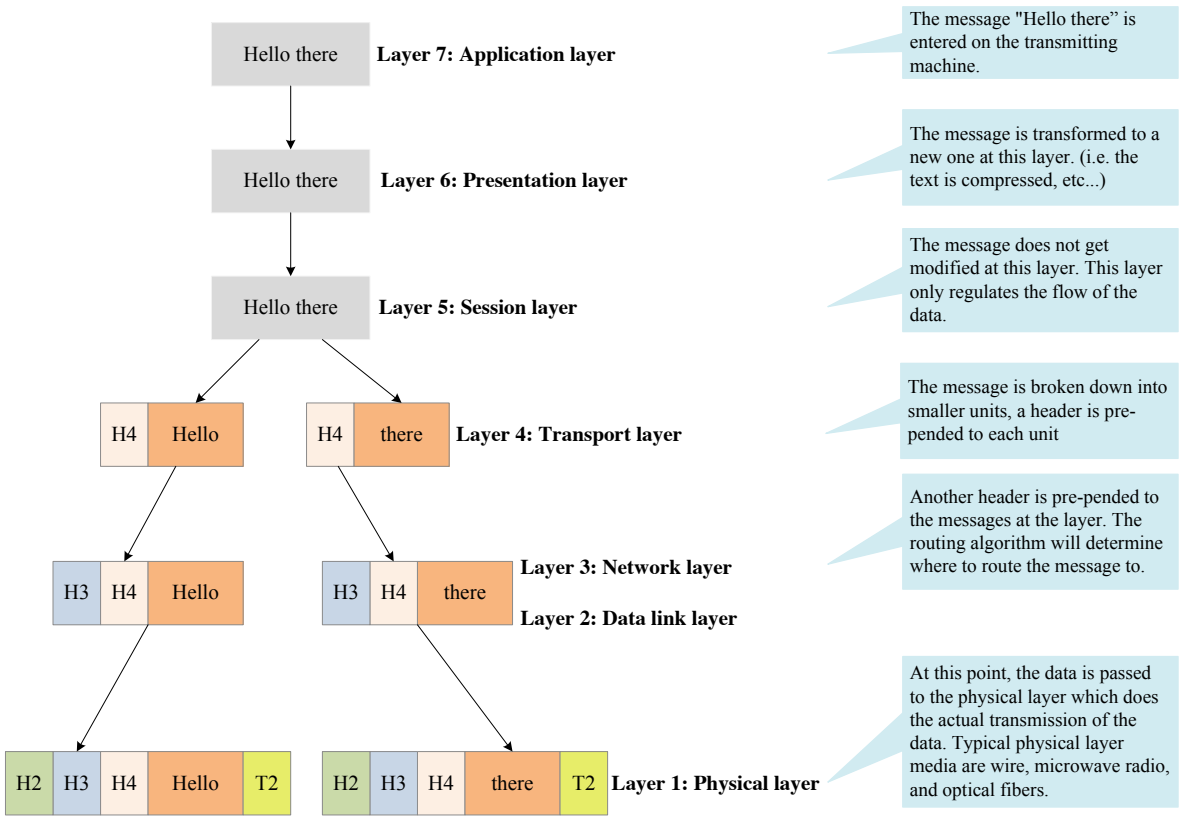



Figure 1.2: Transmitting machine

## 1.3 TCP/IP Architecture

- Transmission Control Protocol (TCP) and Internet Protocol (IP).
- TCP/IP protocols map to a four-layered conceptual model: Application, Transport, Internet, and Network Interface.
- TCP/IP is the foundation of the Internet.

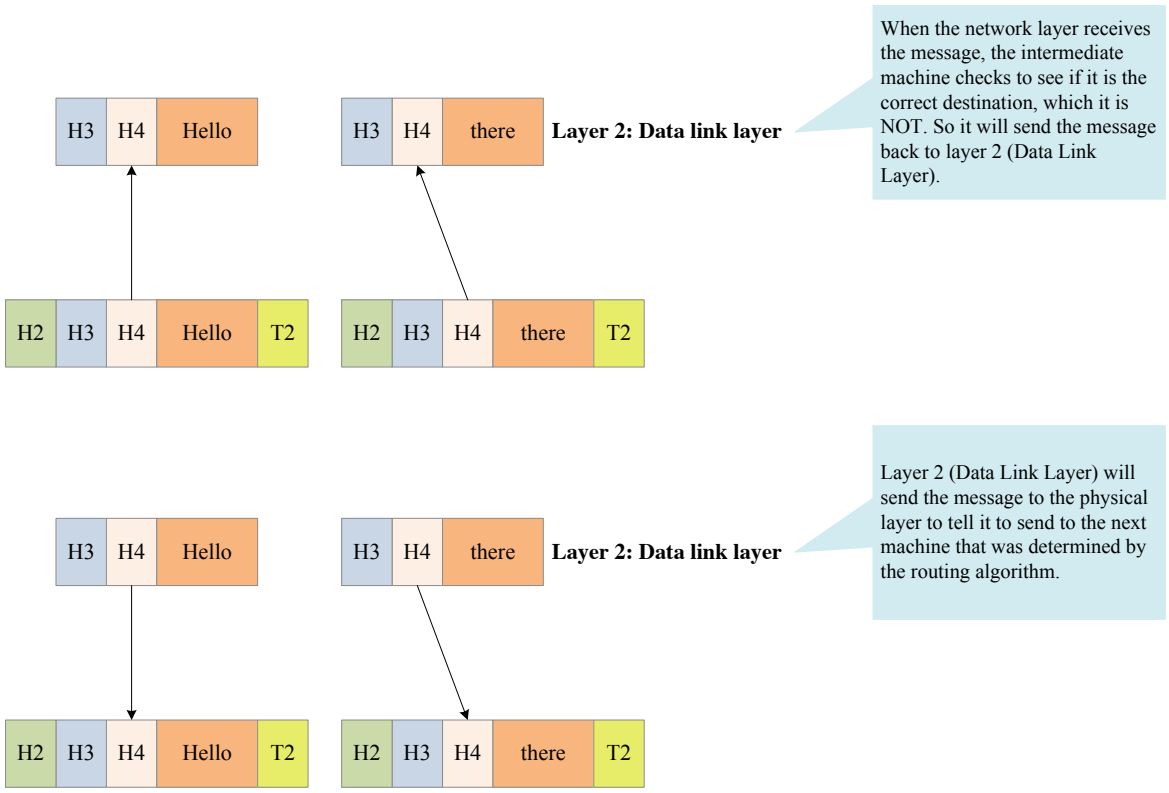



Figure 1.3: Intermediate machine

- A set of protocols allowing communication across diverse networks
- This model is officially known as the *TCP/IP Internet Protocol Suite* but is often referred to as the *TCP/IP protocol family*.

### 1.3.1 Application Layer

- Provide services that can be used by other applications.
- Contains all protocols and methods that fall into the area of process-to-process communications across an Internet Protocol (IP) network.
- Incorporate the functions of top 3 OSI layers.

Examples: Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), Domain Name System (DNS) and Real-time Transport Protocol (RTP).
HTTP protocol, format in request, is the dialogue between client and server. A web page may contain text, graphics, Macromedia Flash objects and perhaps a Java applet . Different files, different downloads, the browser keeps tracks of downloads.

### 1.3.2 Transport Layer

The transport layer or layer 4 provides end-to-end communication services for applications. There exists two kinds of services: TCP & UDP.

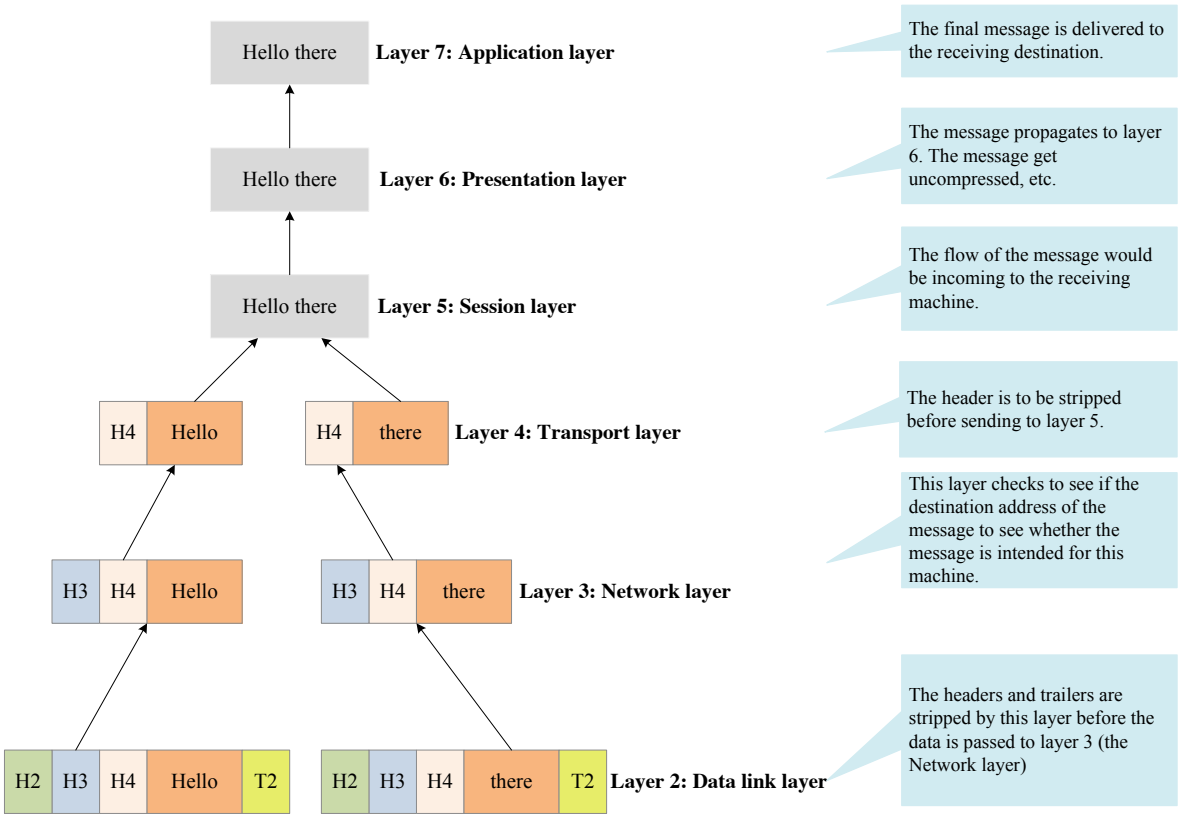



Figure 1.4: Receiving machine

- TCP *Transmission Control Protocol*, reliable connect-oriented transfer of a byte stream.
- UDP *User Datagram Protocol*, best-effort connectionless transfer of individual messages.

The applications at any given network address are distinguished by their TCP or UDP port.

- In computer networking, a port is an application-specific software construct serving as a communications endpoint in a computer's host operating system.
- The protocols that primarily use the ports are the Transport Layer protocols, (such TCP and UDP).
- A port is identified for each address and protocol by a 16-bit number, commonly known as the port number.
- The port number completes the destination address for a communications session.
- Use in URLs: Port numbers can be seen in the URL of a website or other services. By default, HTTP uses port 80 and HTTPS uses port 443, but a URL like *http://www.example.com:8000/path/* specifies that the web site is served by the HTTP server on port 8000.

The port numbers are divided into three ranges:

- *Well-known ports:* Are those from 0 through 1023. Examples are:
  21: (FTP)25: (SMTP), 53: (DNS) service, 80: (HTTP) used in the World Wide Web, etc.

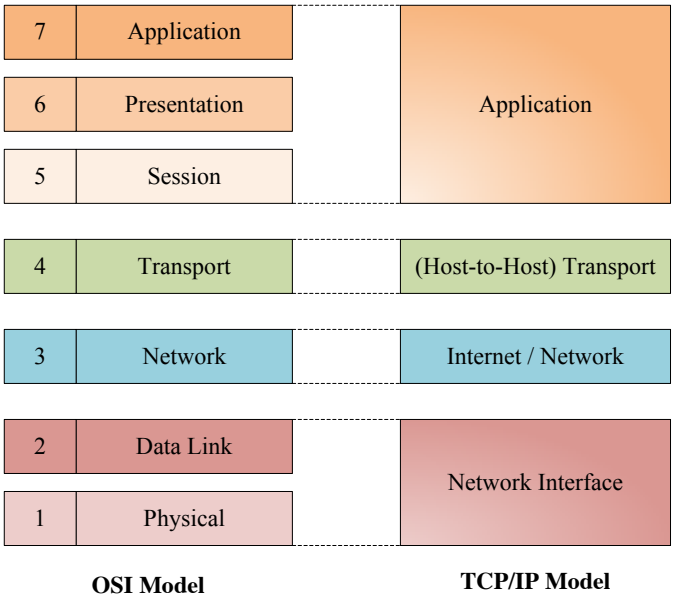



Figure 1.5: TCP/IP Architecture

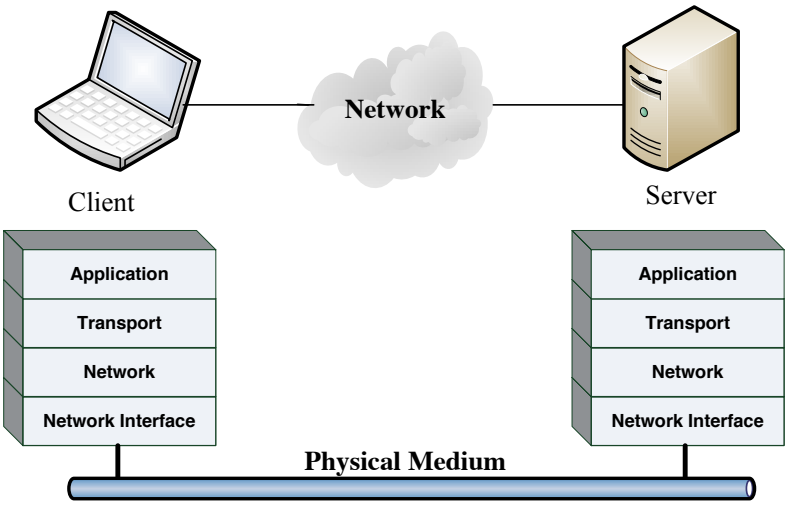



Figure 1.6: TCP/IP Example

- *Registered ports:* The registered ports are those from 1024 through 49151.
- *Dynamic or private ports:* are those from 49152 through 65535.

### 1.3.3 Internet/ Network - Internetworking Layer

The Internet Layer solves the problem of sending packets across one or more networks. It requires sending data from the source network to the destination network. This process is called routing. In the Internet Protocol Suite(IP), the Internet Protocol performs two basic functions:

- *Host addressing and identification:* This is accomplished with a hierarchical addressing system (IP address).
- *Packet routing:* This is the basic task of getting packets of data (datagrams) from source

to destination by sending them to the next network node (router) closer to the final destination.

An **Internet Protocol address** (IP address) is a numerical label assigned to each device (e.g., computer, printer) participating in a computer network that uses the Internet Protocol for communication.

The designers of the Internet Protocol defined :

- *Internet Protocol Version 4 (IPv4):* An IP address as a 32-bit number.
- *Internet Protocol Version 6 (Ipv6):* use 128 bits for the address.

IP addresses are binary numbers, but they are usually stored in text files and displayed in human-readable notations, such as 172.16.254.1 (for IPv4), and 2001:db8:0:1234:0:567:8:1 (for IPv6).

### 1.3.4 Network Interface Layer Link Layer

- Is the networking scope of the local network connection to which a host is attached.
- Is used to move packets between the Internet Layer interfaces of two different hosts on the same link.
- This regime is called the *link* in Internet literature.
- This is the lowest component layer of the Internet protocols, as TCP/IP is designed to be hardware independent.
- As a result TCP/IP is able to be implemented on top of virtually any hardware networking technology.

## 1.4 Example: HTTP Dialogue (Web server)

First of all, we have to mention that the server is determined by its address IP and the application is determined by the port number TCP/UDP.

### 1.4.1 Successive Encapsulation of Descending Application Data

### 1.4.2 Communication with connected mode (TCP)

1. The client sends a sequence of synchronization with a sequence number
2. The server responds with an acceptance (acknowledgment) and a sequence of synchronization with a sequence number.
3. The client responds with an acceptance (acknowledgment).
4. The client sends the data.
5. The server responds with an acceptance (acknowledgment).
6. The server sends the data.

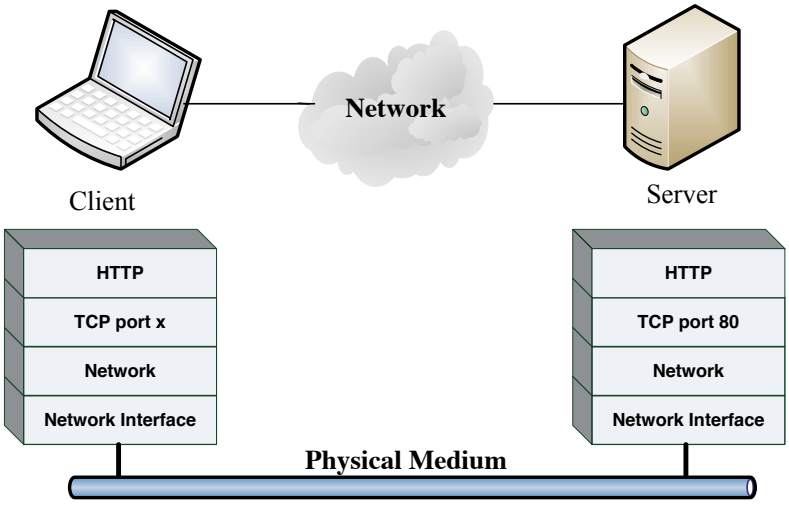



Figure 1.7: HTTP Dialogue (Web server)

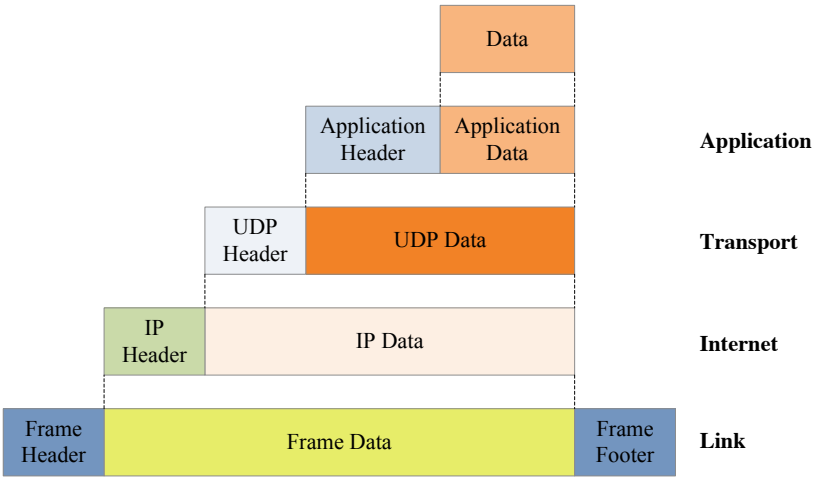



Figure 1.8: Successive Encapsulation of Descending Application Data

7. The client responds with an acceptance (acknowledgment).
8. 
9. The client asks to close the session.
10. The server responds with an acceptance (acknowledgment) .

## 1.5 IP Packet Structure

An IP packet consists of a header section and a data section.

### 1.5.1 IP Header

The IPv4 packet header consists of 14 fields, of which 13 are required. The $14^{th}$ field is optional and aptly named: options.
For example (see figure 1.10)

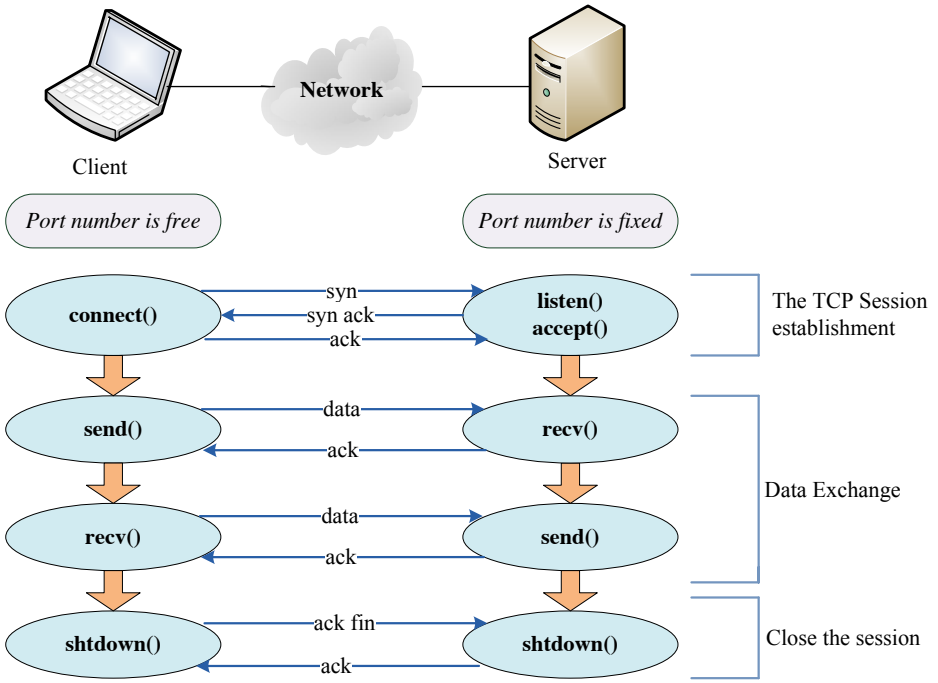



Figure 1.9: Communication with connected mode (TCP)

| bit offset | 0 – 3 | 4 – 7 | 8 – 13 | 14 – 15 | 16 – 18 | 19 – 31 |
|---|---|---|---|---|---|---|
| 0 | Version | Header Length | Differential Services Code Point | Explicit Congestion Notification | Total Length | |
| 32 | Identification | | | | Flags | Fragment Offset |
| 64 | Time to Live | | Protocol | | Header Checksum | |
| 96 | Source IP Address | | | | | |
| 128 | Destination IP Address | | | | | |
| 160 | Options (if Header Length > 5) | | | | | |
| 160 or 192+ | Data | | | | | |

Figure 1.10: IP Header

- **Version:** For IPv4, this has a value of 4 (hence the name IPv4).
- **Header Length:** this field specifies the size of the header. The minimum value for this field is 5.
- **Differentiated Services Code Point (DSCP):** Originally defined as the Type of Service field. An example is Voice over IP (VoIP).
- **Explicit Congestion Notification (ECN):** is an optional feature that is only used when both endpoints support it and are willing to use it. Congestion may be handled only by the transmitter. It must also be an echo of the congestion indication by the receiver to the transmitter.
- **Total Length:** defines the entire datagram size, including header and data, in bytes. The minimum-length datagram is 20 bytes (20-byte header + 0 bytes data) and the maximum is $65,535$ bytes the maximum value of a $16-bit$ word.

- **Identification:** This field is an identification field and is primarily used for uniquely identifying fragments of an original IP datagram.
- **Flags:** is used to control or identify fragments. They are (in order, from high order to low order):
  - bit 0: Reserved; must be zero.
  - bit 1: Don't Fragment (DF).
  - bit 2: More Fragments (MF).
- **Fragment Offset:** This field indicates where in the datagram this fragment belongs. Tells the receiver how to order the fragments measured in units of eight-byte blocks. The first fragment has an offset of zero.
  Examples:
  - If host $H_1$ sends a $1,500$ octet datagram ($20 - octet$ header and $1,480$ octets of data) to host $H_2$, router $R$ will fragment the datagram into two fragments.
  - The first fragment will contain a $20 - octet$ header (FRAGMENT OFFSET is zero and the MORE flag is set) and 976 octets of data (four octets are wasted).
  - The second fragment will contain a $20 - octet$ header (FRAGMENT OFFSET is $976/8 = 122$ and the MORE flag is clear) and the remaining 504 octets of data.
- **Time To Live (TTL):** This field limits a datagram's lifetime. It is specified in seconds, but time intervals less than 1 second are rounded up to 1. Each router that a datagram crosses decrements the TTL field by one. When the TTL field hits zero, the packet is discarded. Typically, a message is sent back to the sender to inform it that the packet has been discarded.
- **Protocol:** This field defines the protocol used. (TCP, SMP)
- **Header Checksum:** is used for error-checking of the header. At each hop, the checksum of the header must be compared to the value of this field. If a header checksum is found to be mismatched, then the packet is discarded.
- **Source address:** An IPv4 address indicating the sender of the packet.
- **Destination address:** An IPv4 address indicating the receiver of the packet.
- **Options:** Additional header fields. Used to specify various TCP options.
- **Data:** data to be send to the receiver.

## 1.6 UDP Packet Structure

The UDP header structure is shown as follows (figure 1.11):

| offset (bits) | 0 – 15 | 16 – 31 |
|---|---|---|
| 0 | Source Port Number | Destination Port Number |
| 32 | Length | Checksum |
| 64+ | Data | |

Figure 1.11: UDP Header

- **Source port:** It is an optional field. When used, it indicates the port of the sending process and may be assumed to be the port to which a reply should be addressed in the absence of any other information. If not used, a value of zero is inserted.
- **Destination port:** This field identifies the receiver's port and is required.
- **Length:** The length in octets of this user datagram, including this header and the data. The minimum length is 8 bytes since that's the length of the header.
- **Checksum:** The checksum field is used for error-checking of the header and data. If no checksum is generated by the transmitter, the field uses the value all-zeros. This field is not optional for IPv6.
- **Data:** UDP data field.

## 1.7 TCP Packet Structure

The TCP header structure is shown as follows (figure 1.12):

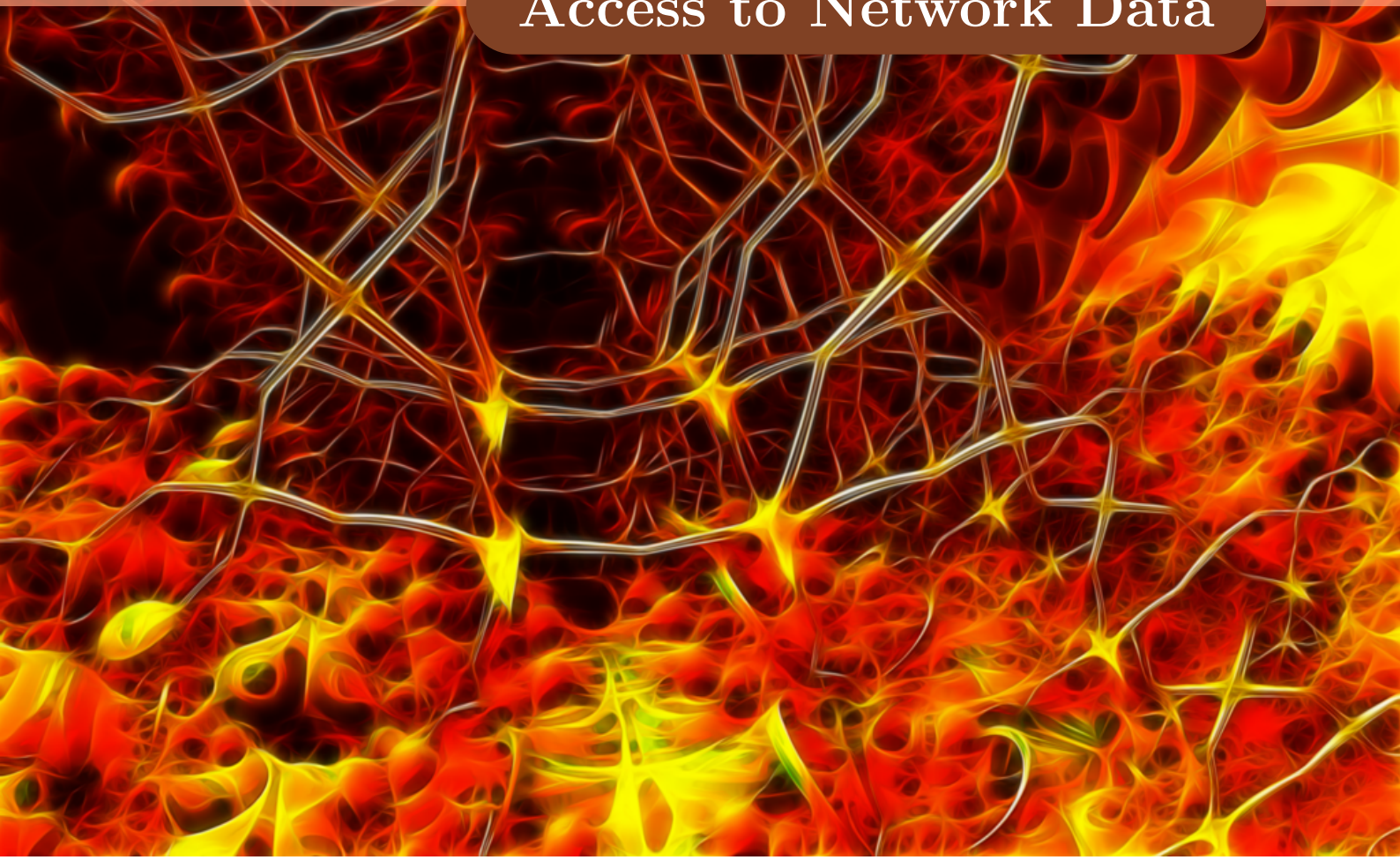
| offset (bits) | 0 – 15 | | | | | | | | 16 – 31 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | Source Port | | | | | | | | Destination Port |
| 32 | Sequence number | | | | | | | | |
| 64 | Acknowledgment number | | | | | | | | |
| 96 | Offset | Resrvd | U | A | P | R | S | F | Window |
| 128 | Checksum | | | | | | | | Urgent pointer |
| 160 | Option + Padding | | | | | | | | |
| 192+ | Data | | | | | | | | |

Figure 1.12: TCP Header

- **Source port:** Source port number.
- **Destination port:** Destination port number.
- **Sequence number:** every octet of data sent over a TCP connection has a sequence number. Since every octet is sequenced, each of them can be acknowledged. the sequence number of the first data octet in this segment (except when SYN is present).
- **Acknowledgment number:** Sequence number of next expected byte
- **Data offset:** 4 bits. The number of $32 - bit$ words in the TCP header, which indicates where the data begins. length of header (the options are part of the header).
- **Reserved:** 6 bits. Reserved for future use. Must be zero.
- **Control bits:** 6 bits. The control bits may be (from right to left):
  - U (URG) Urgent pointer field significant.
  - A (ACK) Acknowledgment field significant.
  - P (PSH) Push function. Asks to push the buffered data to the receiving application. Data to be sent immediatel
  - R (RST) Reset the connection.
  - S (SYN) Synchronize sequence numbers.
  - F (FIN) No more data from sender.
  - **Window:** the size of the receive window, which specifies the number of bytes (beyond the sequence number in the acknowledgment field) that the receiver is currently willing to receive.
  - **Checksum** The 16-bit checksum field is used for error-checking of the header and data Checksum calculated on the full TCP header and data.
  - **Urgent Pointer:** if the URG flag is set, present relative position of last urgent data.
  - **Options Option:** may be transmitted at the end of the TCP header. All options are included in the checksum. An option may begin on any octet boundary.
  - **Data:** TCP data or higher layer protocol.

# Chapter 2

# Access to Network Data



## 2.1 Hosts Access functions

### 2.1.1 Structure "hostent"

The **hostent** structure is used by functions to store information about a given host, such as host name, IPv4 address, and so forth. An application should never attempt to modify this structure or to free any of its components.
It is defined on < *netdb.h* > as:

Listing 2.1: Structure "hostent"

```
1 struct hostent {
2         char *h_name;           /* official name of host */
3         char **h_aliases;       /* alias list */
4         int h_addrtype;         /* host address type */
5         int h_length;           /* length of address */
6         char **h_addr_list;     /* list of addresses */
7 }
```

- ***h_name:** The official name of the host (PC)

- ****h_aliases:** A NULL-terminated array of alternate names.
- **h_addrtype:** The type of address being returned (always AF_INET).
- **h_length:** The length, in bytes, of each address.
- ****h_addr_list:** A NULL-terminated list of IP addresses for the host. Addresses are returned in network byte order. The macro h_addr is defined to be h_addr_list[0] for compatibility with older software

### Example 1

To define an instance of hostent:

Listing 2.2: Define an instance of hostent

```
1 struct hostent * host
```

Get the host name:

Listing 2.3: Get the host name

```
1 host->h_name
```

Access to the IP addresses:

Listing 2.4: Access to the IP addresses

```
1 host -> h_addr_list[0][1].h_addr_list[0][2].h_addr_list[0][3].
      h_addr_list[0][4];
```

### Example 2

Indication for the double pointer and 2D array

Listing 2.5: Double pointer and 2D array

```
1 #include <stdio.h>
2 #include <math.h>
3 int main(void)
4 {/* Declare the '2D Array' */
5          char ** ptr = new char * [5];
6          ptr[0] = new char[20];
7          ptr[1] = new char[20];
8          ptr[2] = new char[20];
9          ptr[3] = new char[20];
10         ptr[4] = new char[20];
11         ptr[0] = "Hello ";
12         ptr[1] = "wond";
13         ptr[2] = "er";
14         ptr[3] = "ful";
15         ptr[4] = "world !!!";
```

```
16          printf("%c", *ptr[2] );// printf("%c", ptr[2][0] );
17 }
```

### 2.1.2 Gethost

#### Gethostbyname

The *gethostbyname ()* function returns a structure of type hostent for the given host name (e.g. *gethostbyname (name)*). *name* specifies the name of the host.
For example: www.wikipedia.org, it can be : hostname, or IPv4 address in standard dot notation , or IPv6 address in colon notation.

The syntax of *gethostbyname* is:

Listing 2.6: Syntax of *gethostbyname*

```
1 struct hostent * gethostbyname(const char *name);
```

Get the IP address corresponding to a given host name. The question here is how to find the IP address of a machine (hostname)? This can be by a DNS Server or hosts file.

If the IP address cannot be found. An error occurred. On error, a null pointer is returned. In this case the *h_errno* variable gives the reason for the error:

- HOST_NOT_FOUND: host unknown.
- NO_ADDRESS or NO_DATA: no known IP address.
- NO_RECOVERY: A non-recoverable name server error occurred.
- TRY_AGAIN: Temporary Error

For example:

Listing 2.7: Example of using *gethostbyname*

```
1 struct hostent * hp;
2 hp = gethostbyname("myhost");
3 if (hp == NULL)
4 {
5          printf("gethostbyname() failed\n");
6 }
7 else
8 {
9          printf("%s = ", hp->h_name);
10 }
```

#### Gethostbyaddr

The **gethostbyaddr()** function returns a structure of type hostent for the given host address *addr* of length *len* and address type type. The only valid address type is currently **AF_INET**.

Where:
**AF_INET** is Internet Protocols.
**Len** is the length of the ip address in bytes.
The syntax of **gethostbyaddr()** is:

Listing 2.8: Syntax of *gethostbyaddr()*

```
1 ptr = gethostbyaddr (char *addr,int len,int type );
```

For example:

Listing 2.9: Example of using *gethostbyaddr*

```
1 char ent[20];
2 struct hostent * host;
3 int p;
4 printf("Enter the name: \n");
5 scanf("%s",ent);
6 //can give the IP address
7 printf("Enter the address : %s \n",ent);                    // "127.0.0.1"
8 host = gethostbyaddr(ent,4,AF_INET);
```

**Read the hosts file:**

- **Sethostent:** Open the hosts file / access to the DNS server and places a pointer to the first host
- **Gethostent:** give the next host.
- **Endhostent:** close the hosts file/ access to DNS server.

**Function declaration:**

- *#include < netdb.h >*
- struct hostent *gethostbyname(const char *name);
- struct hostent *gethostbyaddr(const char *addr, int len, int type);
- void sethostent(int stayopen);
- struct hostent *gethostent();
- void endhostent(void);

**Example:** Reading of the entire host defined in the hosts file.

Listing 2.10: Reading of the entire host defined in the hosts file

```
#define LMAX 1024
#include <stdio.h>
#include <netdb.h>
int main()
{
struct hostent * host;
sethostent (1);
printf("List of known hosts:\n");
while ((host = gethostent()) != NULL)
        printf(" name : %s \n",host->h_name);
endhostent();
}
```

Modify the first example to display also the IP address of the hosts

Listing 2.11: Modify the first example to display also the IP address of the hosts

```
#define LMAX 1024
#include <stdio.h>
#include <netdb.h>
int main()
{
// Example: Read all the defined hosts in the file hosts
struct hostent * host;
sethostent (1);
printf("List of known hosts:\n");
while ((host = gethostent()) != NULL)
{
        printf(" nom : %s \n",host->h_name);
        printf(" IP: %x", host->h_addr_list[0][0]);
        printf(".");
        printf("%x", host->h_addr_list[0][1]);
        printf(".");
        printf("%x", host->h_addr_list[0][2]);
        printf(".");
        printf("%x", host->h_addr_list[0][3]);
}
endhostent();
}
```

## 2.2 Services Access functions

In this section, we need to answer the following 2 questions:

1. How to find the port associated with a service number?

2. How to find if a port number is available?

### 2.2.1 Structure "servent"

It is defined in < *netdb.h* >:

Listing 2.12: Structure "servent"

```
1 struct servent
2 {
3         char *s_name;             /* official service name */
4         char **s_aliases;         /* alias list */
5         int s_port;                       /* port number */
6         char *s_proto;            /* protocol to use */
7 }
```

The members of the **servent** structure are:

- **s_name:** The official name of the service.
- **s_aliases:** A zero terminated list of alternative names for the service.
- **s_port:** the port number for the service given in network byte order.
- **s_proto:** The name of the protocol to use with this service.

The syntax of **servent** is:

Listing 2.13: Syntax of *servent*

```
1 struct servent *app
```

### 2.2.2 Getserv Functions

#### Getservbyname

**Getservbyname** performs a research of a service by name and protocol. It returns a *servent* structure that matches the service *name* using protocol *proto*. If proto is NULL, any protocol will be matched. A NULL pointer is returned if an error occurs or the end of the file is reached. The syntax of **getservbyname** is:

Listing 2.14: Syntax of *getservbyname*

```
1 struct servent *getservbyname(const char *name, const char *proto)
```

Where:
**name:** service name (string end by 0 binary).
**proto:** protocol name to use ("tcp", "udp" or "").

For example:

Listing 2.15: Example of using *getservbyname*

```
#include<netdb.h>
struct servent *getservbyname(const char *servname, const char *
    protoname);
struct servent *sptr;
sptr = getservbyname("domain", "udp");  /* DNS using UDP */
sptr = getservbyname("ftp", "tcp");              /* FTP using TCP */
sptr = getservbyname("ftp", NULL);               /* FTP using TCP */
sptr = getservbyname("ftp", "udp");              /* This call will fail
    */
```

Another example:

Listing 2.16: Example of using *getservbyname*

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

struct servent *appl_name;
char name[4] = "FTP";
char proto[4] = "TCP";
int port;
appl_name = getservbyname(name, proto);
if (!appl_name)
        printf("unknown application %s\n", name);
else
{
        port = appl_name->s_port;
        printf("getservbyname was successful\n");
}
```

### Getservbyport

**Getservbyport** performs a research of a service by port number and protocol. It returns a *servent* structure that matches the port *port* (given in network byte order). If *proto* is NULL, any protocol will be matched.
The syntax of **getservbyport** is:

Listing 2.17: Syntax of *getservbyport*

```
struct servent *getservbyport(int port, const char *proto);
```

Where:

**port:** port number.
**proto:** protocol name to use (”tcp”, ”udp” or ””).

For example:

Listing 2.18: Example of using *getservbyport*

```
#include<netdb.h>
struct servent *getservbyport(int port, const char *protoname);
struct servent *sptr;
sptr = getservbyport(htons(53), "udp");     /* DNS using UDP */
sptr = getservbyport(htons(21), "tcp");     /* FTP using TCP */
sptr = getservbyport(htons(21), NULL);      /* FTP using TCP */
sptr = getservbyport(htons(21), "udp");     /* This call will fail */
```

The **htons()** function converts the unsigned short integer *hostshort* from host byte order to network byte order.

Listing 2.19: Example of using *getservbyport*

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
struct servent *appl_name;
int port;
char proto[4] = "TCP";
char *name; port = 21;
appl_name = getservbyport(htons(port), proto);
if (!appl_name)
        printf("unknown application %s\n", name);
else
{
        name = appl_name->s_ name;
        printf("getservbyport was successful\n");
}
```

### 2.2.3 Reading the Hosts File

**setservent:**

- Opens a connection to the /etc/services file.
- Sets the next entry to the first line in the service file.

Syntax:

Listing 2.20: Syntax of using *setservent*

```
void setservent(int stayopen);
```

If *stayopen* is true (1), then the file will not be closed between servent calls (**getservbyname()**, **getservbyport()**).

Example:

Listing 2.21: Example of using *setservent*

```
1 setservent(0);
```

**getservent:**

- Reads a service from a services file.
- Returns a **servent** structure.

Syntax:

Listing 2.22: Syntax of using *getservent*

```
1 struct servent *getservent(void);
```

Example:

Listing 2.23: Example of using *getservent*

```
1 struct servent * serv;
2 serv = getservent()
```

**endservent**

- Close the services file.

Syntax:

Listing 2.24: Syntax of using *endservent*

```
1 void endservent(void);
```

Example:

Listing 2.25: Example of using *endservent*

```
1 endservent ();
```

## 2.3 The Conversion Function

It consists of IP addresses manipulating functions.

### 2.3.1 inet_aton

**inet_aton** function converts the specified string, from the Internet standard dot notation (pointed decimal format or IPv4 numbers-and-dots notation), into binary format in the network byte order. The address is returned in an *in_addr* structure pointed by the second parameter of the function.
The structure *in_addr* is defined in *netinet/in.h*:

Listing 2.26: *in_addr* structure

```
1 struct in_addr
2 {
3         unsigned long int s_addr;
4 }
```

The **inet_aton()** function returns 1 if the address is successfully converted, or 0 if the conversion failed or in the error case.
The syntax of **inet_aton()** is:

Listing 2.27: Syntax of *getservbyport*

```
1 int inet_aton(const char *cp, struct in_addr *inp);
```

Where:

- *cp:* Points to a string in Internet standard dot notation.
- *inp:* Buffer where the converted address is to be stored.

The values specified using dot notation take one of the following forms (cp):

| | |
|---|---|
| a.b.c.d | When four parts are specified, each is interpreted as a byte of data |
| a.b.c | When a three-part address is specified, the last part is interpreted as a 16-bit |
| a.b | When a two-part address is supplied, the last part is interpreted as a 24-bit |
| A | When only one part is given, the value is stored directly in the network address without any byte rearrangement. |

**Example:**

Listing 2.28: Example of using *inet_aton*

```
1 struct in_addr adresse;
2 if ( inet_aton("187.43.32.1", & adresse) == 0 )
3         perror("inet_aton() failed");
```

### 2.3.2 inet_ntoa

The **inet_ntoa** function performs the conversion from binary to decimal pointed format. It converts the Internet host address, given in network byte order, to a string in IPv4 dotted-decimal notation. The string is returned in a statically allocated buffer, which subsequent calls will overwrite.
The syntax of **inet_ntoa** is:

Listing 2.29: Syntax of *inet_ntoa*

```
1 char *inet_ntoa(struct in_addr in);
```

**Example:**

Listing 2.30: Example of using *inet_ntoa*

```
1 #include <stdio.h>
2 #include <netdb.h>
3 #include <sys/types.h>
4 #include <netinet/in.h>
5 #include <arpa/inet.h>
6 void main(void)
7 {
8 hostent * record = gethostbyname("rabbit.eng.miami.edu");
9 if (record==NULL)
10 {
11         printf("gethostbyname failed");
12 }
13 else
14 {
15         in_addr * address=(in_addr * )record->h_addr;
16         printf("IP Address: %s\n", inet_ntoa(* address));
17 }
18 }
```

Listing 2.31: Example of using *inet_aton* and *inet_ntoa*

```
1 $ ./a.out 226.000.000.037       # Last byte is in octal
2 226.0.0.31
3 $ ./a.out 0x7f.1                # First byte is in hex
4 127.0.0.1
```

### 2.3.3 inet_addr

**inet_addr** performs the conversion from the decimal dotted format to the hexadecimal format.

### 2.3.4 Htonl

**Htonl** hosts to network long (4 bytes). It converts an IP address in the computers integer format to the standard network format machine.

**Htonl** is included in $\#include < sys/param.h >$.

### 2.3.5 htons

**htons** hosts to network short (2 bytes).

### 2.3.6 ntohl

Network to host long (4 bytes).

### 2.3.7 ntohs

Network to host short (2 bytes). 256 in the network $\rightarrow$ 1 in the host.

# Exercises

## Exercise 1

Write a program that allows the user to enter the name of the host and give as output the host name and all the aliases and the ip @.

**Answer of exercise 1**

Listing 2.32: Solution of Exercises 2.1

```
#include <stdio.h>
#include <string.h>
#include <netdb.h>
int main()
{
        char ent[20];
        struct hostent * host;
        int ind,p;
        printf("Enter the name of the machine:\n");
        scanf("%s",ent);
        host = gethostbyname(ent);
        if (host != NULL)
        {
                printf(" Host name is: %s (",host->h_name);
                for (p=0;host->h_aliases[p]!=NULL;p++)
                        printf(" %s", host->h_aliases[p]);
                printf(") ");

                for (p=0;p<host->h_length;p++)
                {
                        if (p==host->h_length-1)
                                printf("%x\n",host->h_addr_list[0][p]);
                        else
                                printf("%x.",host->h_addr_list[0][p]);
                }
        }
        else
                printf("Unknown Host \n");
}
```

The execution of this program is:

```
Enter the name of the machine: mylinux
Host name:
mylinux (localhost.localdomain localhost) 7f.0.0.1
```

## Exercise 2

Write a program that allows reading the services file and displaying all the services list.

**Answer of exercise 2**

Listing 2.33: Solution of Exercises 2.2

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <netdb.h>
4 int main()
5 {
6 struct servent * serv;
7 setservent (1);
8 printf("services list :\n");
9 serv = getservent()
10 while (serv != NULL)
11         printf(" port : %d name : %s protocol : %s \n",ntohs(serv->
               s_port),serv->s_name,serv->s_proto);
12 endservent();
13 }
```

## Exercise 3

Write a program allows us to enter the service name, and return the service description (port number, name and protocol) for the specific protocols (TCP and UDP), using **getservbyname** function.

### Answer of exercise 3

Listing 2.34: Solution of Exercises 2.3

```
#include <stdio.h>
#include <string.h>
#include <netdb.h>
int main()
{
char ent[20];
struct servent * servtcp;
struct servent * servudp;
printf("Service name :\n");
scanf("%s",ent);
servtcp = getservbyname(ent,"tcp");
if (servtcp != NULL)
        printf(" port : %d name : %s protocol : %s \n",

ntohs(servtcp->s_port),servtcp->s_name,servtcp->s_proto);
else
        printf(" port or name does not exist for TCP \n");
servudp = getservbyname(ent,"udp");
if (servudp != NULL)
        printf(" port : %d name : %s protocol : %s \n",
        ntohs(servudp->s_port),servudp->s_name,servudp->s_proto);
else
        printf("port or name does not exist for UDP \n");
}
```

## Exercise 4

Write a program that allows us to enter the port number, and return the service description (port number, name and protocol) for the specific protocols (TCP and UDP), using **getservbyport** function.

### Answer of exercise 4

Listing 2.35: Solution of Exercises 2.4

```
#include <stdio.h>
#include <string.h>
#include <netdb.h>
int main()
{
struct servent * servtcp;
struct servent * servudp;
int port;
printf("Enter the port number:\n");
scanf("%d",port);
servtcp = getservbyport(htons(port),"tcp");
if (servtcp != NULL)
        printf(" port : %d name : %s protocol : %s \n",ntohs(servtcp->
           s_port),servtcp->s_name,servtcp->s_proto);
                else
        printf("port or name does not exist for TCP \n");
        servudp = getservbyport(htons(port),"udp");
if (servudp != NULL)
        printf(" port : %d name : %s protocol : %s \n",ntohs(servudp->
           s_port),servudp->s_name,servudp->s_proto);
                else
        printf(" port or name does not exist for UDP \n");
}
```

## Exercise 5

Write a program that allows us to enter the IP address in doted-decimal notation, test if the entry address is valid, and display it in the network byte order.
Example:

Enter the IP address in doted decimal notation: 192.10.2.2
**Output:**
inet_aton(192.10.2.2): C00A0202

### Answer of exercise 5

Listing 2.36: Solution of Exercises 2.5

```
#include <stdio.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main()
{
struct in_addr address;
char adr[128];
printf("Enter the IP address in doted-decimal notation:\n");
scanf("%s",adr);
printf("inet_aton (%s) : ",adr);
if (inet_aton(adr,&address)==0)
        printf("invalid Address \n");
else
        printf("%x \n",ntohl(address.s_addr));
return 0;
}
```

## Exercise 6

Write a program that allow:

1. Enter an IP address in hexadecimal format by making the following controls:
   (a) The size should be 8 characters.
   (b) The entered characters must be valid (A-F and 0-9).
2. Display the address in dotted decimal format.

**Topics:**

*Conversion functions of binary IP address to pointed decimal:*

- inet_addr: conversion from pointed decimal format to hexadecimal format.
- inet_ntoa: conversion from hexadecimal format to pointed decimal format.
- inet_aton: conversion from pointed decimal format to hexadecimal format.

*Order functions of memory bytes:*

- htonl: host to network long (4 octets).
- htons: host to network short (2 octets).
- ntohl: network to host long (4 octets).
- ntohs: network to host short (2 octets).

### Answer of exercise 6

Listing 2.37: Solution of Exercises 2.6

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <netinet/in.h>
4 #include <arpa/inet.h>
5 int main()
6 {
7 struct in_addr adresse;
8 char adr[128];
9 unsigned int j,k,k2;
10 unsigned long int p;
11 printf("Enter an IP in Hexadecimal form:\n");
12 scanf("%s",adr);
13 if (strlen(adr)!=8)
14         printf("Enter an address with 8 digits\n") ;
15
16 else
17 {
18         k=0;
19         p=0;
20         for (j=0;j<strlen(adr);j++)
21         {
22                 if (!( ((adr[j]>='0')&&(adr[j]<='9'))||((adr[j]>='a')
                    &&(adr[j]<='f'))||((adr[j]>='A')&&(adr[j]<='F')) ) )
23                      k=1;
24         }
```

```
25
26          if (k==1)
27                  printf("Invalid address\n");
28          else
29          {
30                  for (j=0;j<strlen(adr);j=j+2)
31                  {
32                          if ((adr[j]<='9')&&(adr[j]>='0'))
33                                  k=adr[j]-48;
34                          else if ((adr[j]>='a')&&(adr[j]<='f'))
35                                  k=adr[j]-97+10;
36                          else if ((adr[j]>='A')&&(adr[j]<='F'))
37                                  k=adr[j]-65+10;
38                          if ((adr[j+1]<='9')&&(adr[j+1]>='0'))
39                                  k=k*16+adr[j+1]-48;
40                          else if ((adr[j+1]>='a')&&(adr[j+1]<='f'))
41                                  k=k*16+adr[j+1]-97+10;
42                          else if ((adr[j+1]>='A')&&(adr[j+1]<='F'))
43                                  k=k*16+adr[j+1]-65+10;
44                          p = p * 256 + k;
45                  }
46                  adresse.s_addr = htonl(p);
47                  printf("inet_ntoa before calling inet_ntoa (%s) %s \n",
48                  adr,inet_ntoa(adresse));
49          }
50 }
51 return 0;
52 }
```

**Numerical Example:**

Enter an IP in Hexadecimal form:
**A3b58CDd**

For $j = 0$, $adr[j] = A$. Its ASCII value is 65.

$\Rightarrow k = adr[j] - 65 + 10 = 65 - 65 + 10 = 10$

For $j = j + 1 = 1$, $adr[j] = 3$. Its ASCII value is 51.

$\Rightarrow k = 16 \times k + adr[j + 1] - 48 = 10 \times 16 + 51 - 48 = 163$

$\Rightarrow p = 256 \times p + k = 0 + 163 = \mathbf{163}$ (2.1)

Now, $j = 2$, $adr[j] = b$. Its ASCII value is 98.

$\Rightarrow k = adr[j] - 97 + 10 = 98 - 97 + 10 = 11$

For $j = j + 1 = 3$, $adr[j] = 5$. Its ASCII value is 53.

$\Rightarrow k = 16 \times k + adr[j + 1] - 48 = 11 \times 16 + 53 - 48 = \mathbf{181}$

$\Rightarrow p = 256 \times p + k = 256 \times 163 + 181 = 41909$ (2.2)

$j$ is now incremented by 1 ($j = 4$), $adr[j] = 8$. Its ASCII value is 56.

$\Rightarrow k = adr[j] - 48 = 56 - 48 = 8$

For $j = j + 1 = 5$, $adr[j] = C$. Its ASCII value is 67.

$\Rightarrow k = 16 \times k + adr[j + 1] - 65 + 10 = 8 \times 16 + 67 - 65 = \mathbf{140}$

$$\Rightarrow p = 256 \times p + k = 256 \times 41909 + 140 = 10728844 \tag{2.3}$$

Here, $j = 6$, $adr[j] = D$. Its ASCII value is 68.

$\Rightarrow k = adr[j] - 65 + 10 = 68 - 65 + 10 = 13$

For $j = j + 1 = 7$, $adr[j] = d$. Its ASCII value is 100.

$\Rightarrow k = 16 \times k + adr[j + 1] - 97 + 10 = 13 \times 16 + 100 - 97 + 10 = \mathbf{221}$

$$\Rightarrow p = 256 \times p + k = 256 \times 10728844 + 221 = 2746584285 \tag{2.4}$$

Hence, the final value of $p$ is:

$$p = 2746584285 \tag{2.5}$$

When we convert $p$ to decimal doted notation, we get:

$$p = 2746584285 = 163(256)^3 + 181(256)^2 + 140(256)^1 + 221 \tag{2.6}$$

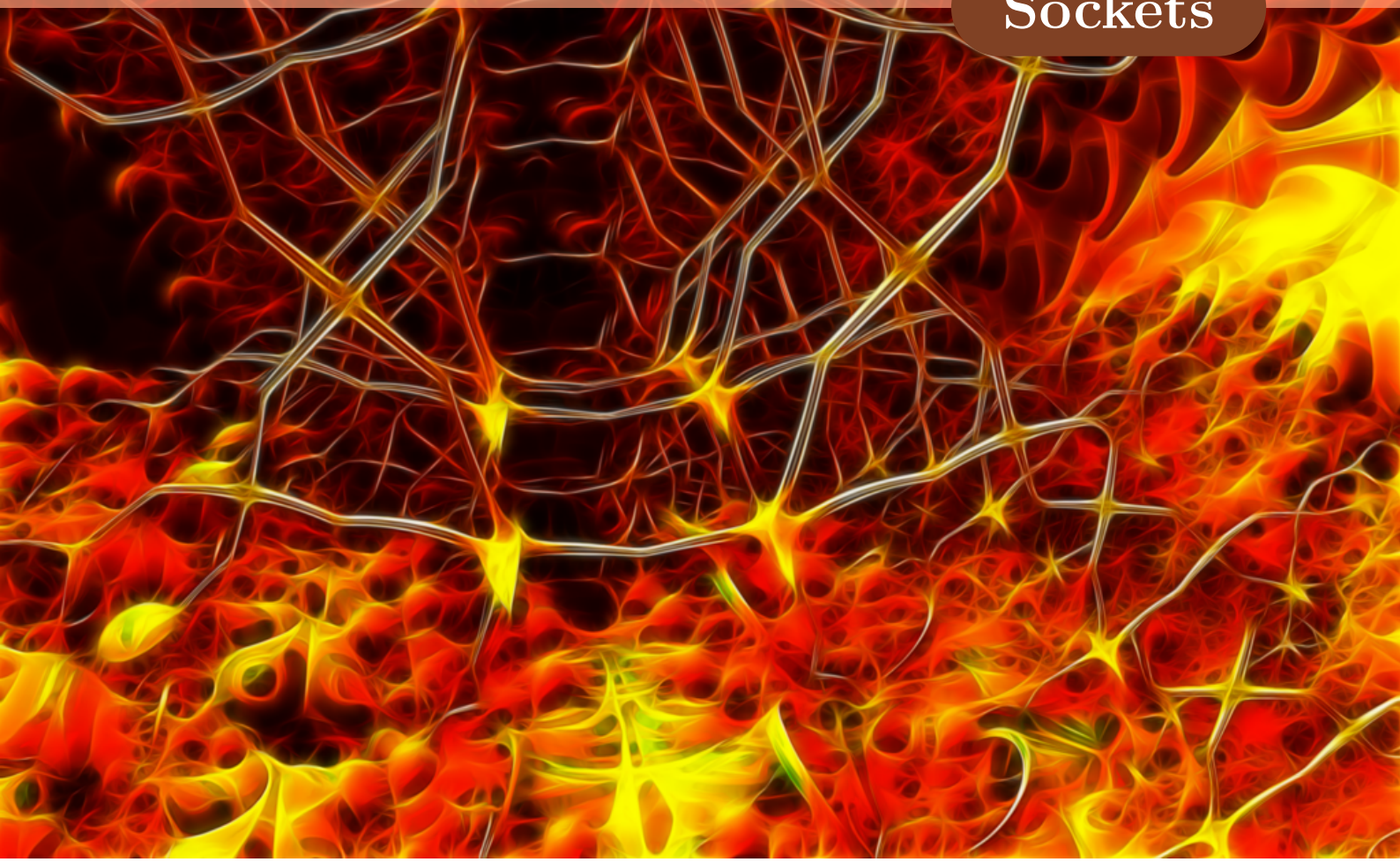
This implies the doted decimal notation of $p$ is equal to 163.181.140.221.

Consequently, the complete display of the program is:

```
Enter an IP in Hexadecimal form:
A3b58CDd
Network to address translation before calling inet_ntoa (A3b58CDd) is: 163.181.140.221
```

# Chapter 3

# Sockets



## 3.1 Concept

A socket is one end-point of a two-way communication link between two programs running on the network. It is an abstraction through which an application may send and receive data. Sockets provide an interface for programming networks at the Layer 3 (network) or 4 (transport). Sockets are also known as the BSD (Berkeley Software Distribution) socket API.

Socket is the standard for communications programming. It allows communication in client / server. Communication interprocess is done through UNIX.
Socket is also an API (application programming interface):

- To code applications performing communication between hosts or between processes.
- Comprises a library for developing applications in the C programming language that perform inter-process communication.
- Providing a set of functions "normalized" to the client / server communication over networks of any type.

A socket is an endpoint of communication to which a name can be bound. A socket has a type and one associated process. Sockets were designed to implement the client-server model for interprocess communication where:

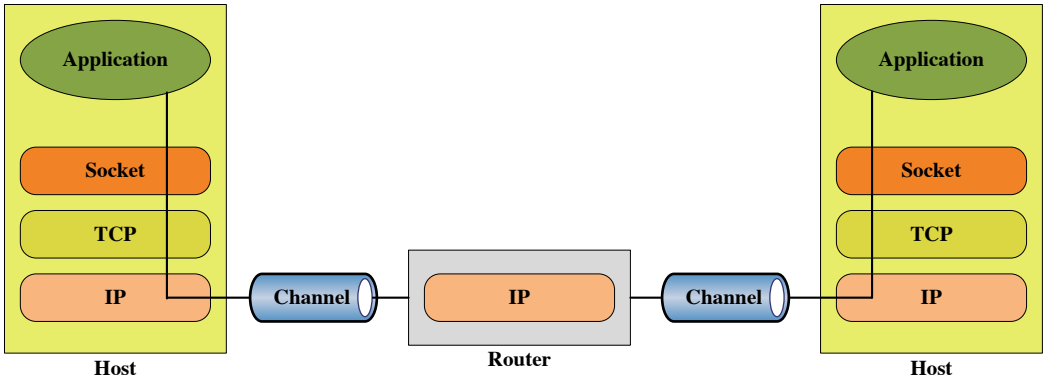



Figure 3.1: Standard API for Networking

- The interface to network protocols needs to accommodate multiple communication protocols, such as TCP/IP, XNS, and UNIX domain.
- The interface to network protocols need to accommodate server code that waits for connections and client code that initiates connections.
- They also need to operate differently, depending on whether communication is connection-oriented or connectionless.
- Application programs may wish to specify the destination address of the datagrams it delivers instead of binding the address with the *open()* call.

To address these issues and others, sockets are designed to accommodate network protocols, while still behaving like UNIX files or devices whenever it makes sense to. Applications create sockets when they need to. Sockets work with the *open()*, *close()*, *read()*, and *write()* system calls.

## 3.2 Socket Operation

### 3.2.1 Socket APIs

- **socket:** creates a socket of a given domain, type, protocol (buy a phone).
- **bind:** assigns a name to the socket (get a telephone number).
- **listen:** specifies the number of pending connections that can be queued for a server socket. (call waiting allowance)
- **accept:** server accepts a connection request from a client. (answer phone)
- **connect:** client requests a connection request to a server (call).
- **send**, **sendto**: write to connection (speak).
- **recv**, **recvfrom**: read from connection (listen).
- **shutdown:** end the call.

### 3.2.2 Communication Connected Mode(TCP)

Server performs the following actions:

- **socket:** create the socket
- **bind:** give the address of the socket on the server
- **listen:** specifies the maximum number of connection requests that can be pending for this process
- **accept:** establish the connection with a specific client
- **send/recv:** stream-based equivalents of read and write (repeated)
- **shutdown:** end reading or writing
- **close:** release data structures

Client performs the following actions:

- **socket:** create the socket
- **connect:** connect to a server
- **send/recv:** (repeated)
- **shutdown**
- **close**

Now, let us consider the following example of communication connected mode by using TCP (figure 3.3)

## 3.3 Socket Structures

### 3.3.1 Struct sockaddr_in

This structure is used for handling internet addresses. The structure used with TCP / IP is AF_INET address (usually the address structures are redefined for each address family). AF_INET addresses using a sockaddr_in structure are defined in $< netinet/in.h >$:
The Syntax of struct sockaddr_in is:

Listing 3.1: struct sockaddr_in

```
#include <netinet/in.h>
struct sockaddr_in
{
        short sin_family;
        unsigned short sin_port;
        struct in_addr sin_addr;
        char sin_zero[8];
};
```

Where:

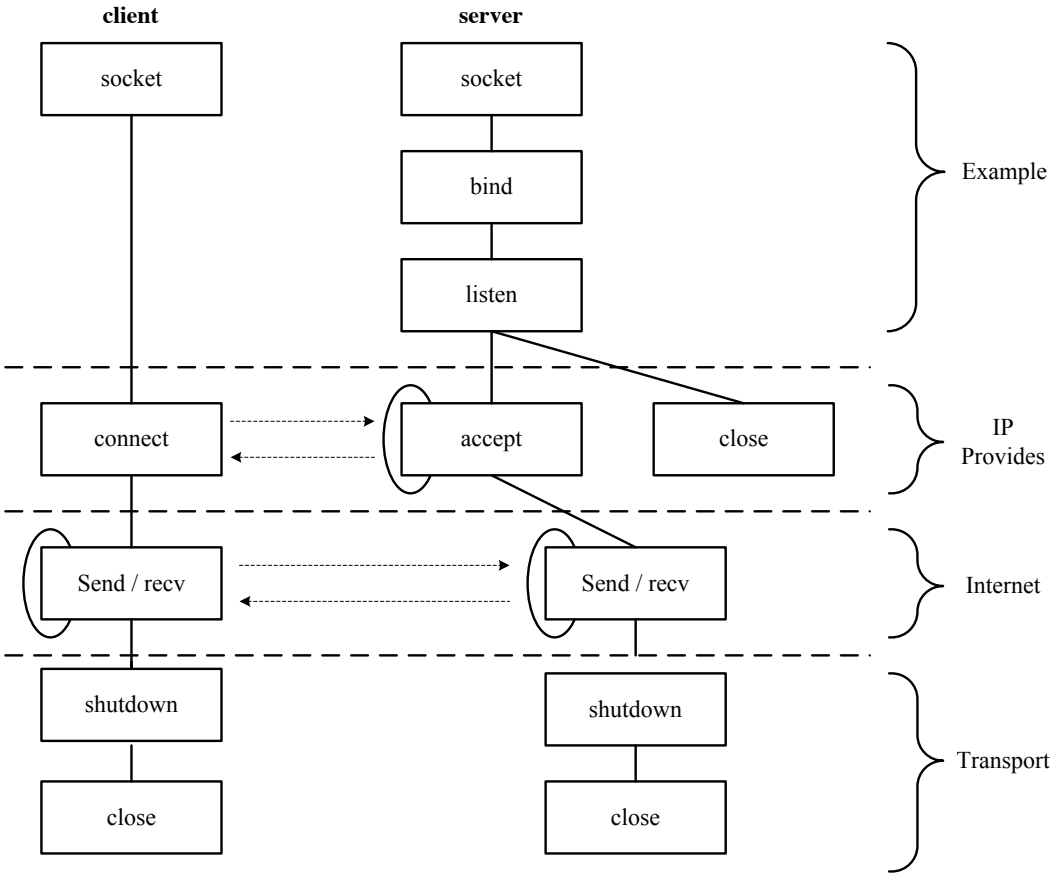



Figure 3.2: Communication Connected Mode(TCP)

- **sin_family:** represents the family type . e.g. AF_INET.
- **sin_port:** represents the port to contact.e.g. htons(3490)
- **sin_addr:** represents the address of the host (in type in_addr).
- **sin_zero[8]:** contains only zeros (since the IP address and port occupy 6 bytes, the remaining 8 bytes must be zero).

These are the basic structures for all functions that deal with internet addresses. They are defined in $< netinet/in.h >$.

**Example:**

Listing 3.2: Example for using struct sockaddr_in

```
1 struct sockaddr_in myaddr;
2 int s;
3 myaddr.sin_family = AF_INET;
4 myaddr.sin_port = htons(3490);
5 inet_aton("63.161.169.137", &myaddr.sin_addr.s_addr);
6 s = socket(PF_INET, SOCK_STREAM, 0);
7 bind(s, (struct sockaddr*)myaddr, sizeof(myaddr));
```

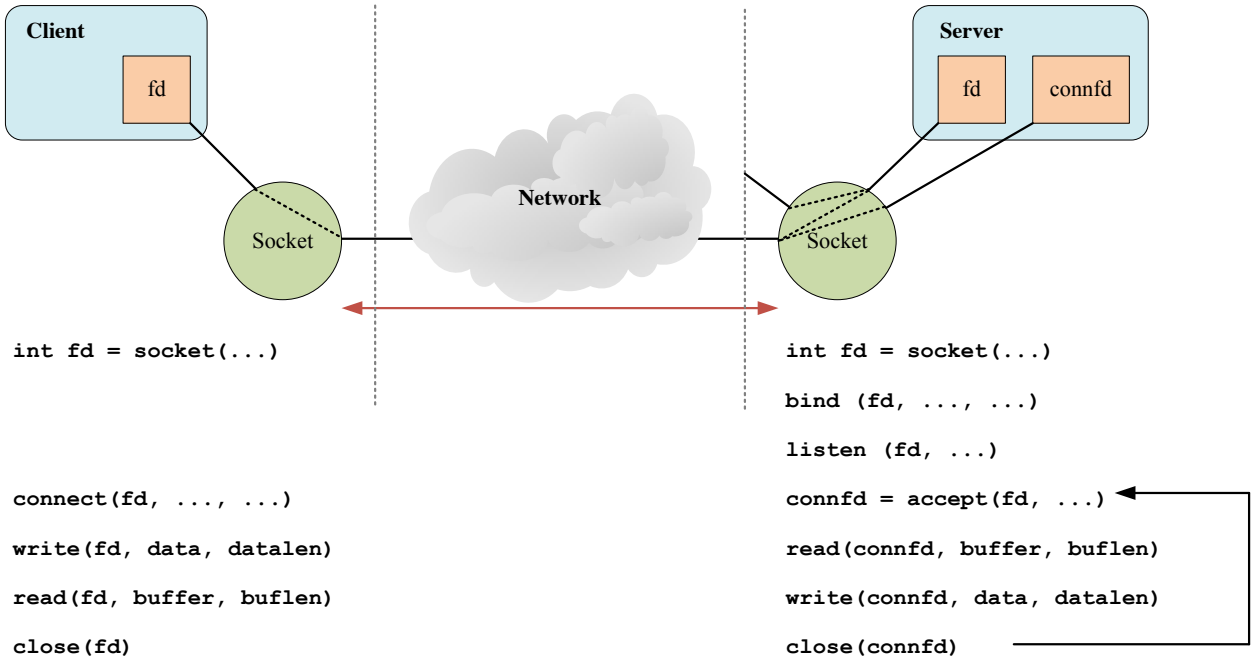



Figure 3.3: Example of communication connected mode by using TCP

### 3.3.2 Struct sockaddr

The address format is highly dependent on the protocol used: the socket interface defines a standard structure (defined in $< sys/socket.h >$). *sockaddr* represents an address:
The Syntax of struct sockaddr is:

Listing 3.3: struct ockaddr

```
1 #include <netinet/in.h>
2 struct sockaddr
3 {
4         u_char sa_len;
5         u_char sa_family;
6         char sa_data[14];
7 };
```

Where:

- **sa_len:** allow to define the address length .
- **sa_ family:** represents the protocol family (AF_INET for TCP/IP).
- **sa_data[14]:** is a string of 14 characters (maximum) containing the address.

### 3.3.3 Struct in_addr

The Syntax of struct in_addr is:

Listing 3.4: struct in_addr

```
struct in_addr
{
        union
        {
                struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
                struct { u_short s_w1,s_w2; } S_un_w;
                u_long S_addr;
        } S_un;
};
```

Where:

- **S_un_b:** Address of the host formatted as four u_chars.
- **S_un_w:** Address of the host formatted as two u_shorts.
- **S_addr:** Address of the host formatted as u_long.

### Union

They are the same as structures, except that, where you would have written struct before, now you write union.

Only one element in the union may have a value set at any given time

Listing 3.5: element in the union

```
struct object {
        char id[20];
        struct coordinates loc;
};
```

Let us modify our structure object from above so that it has a union for indicating dead or alive in it:

Listing 3.6: Union

```
struct object {
        char id[20];
        struct coordinates loc;
        union deadoralive {
                int alive;
                int dead;
        }
};
```

*Note:* Only dead or alive can be set to anything at any one time.

## 3.4 Socket Functions

### 3.4.1 Socket

- Create a socket of a given domain, type, protocol (buy a phone).
- Create an endpoint for communication and returns a descriptor.

**Syntax:**

Listing 3.7: Socket Functions

```
1 int socket(int domain, int type, int protocol);
```

- **domain:** Specifies the communications domain in which a socket is to be created.
  - *AF_UNIX:* UNIX internal protocols.
  - *AF_INET:* Internet Protocols.
  - *AF_NS:* Xerox NS Protocols.
- **type:** Specifies the type of socket to be created. Types defined in $< sys/socket.h >$.
  - *SOCK_STREAM:*
    - Establishes a virtual circuit for stream.
    - Communication( connected mode) on dataflow (TCP).
  - *SOCK_SEQPACKET:*
    - Communication Packet (type x25).
  - *SOCK_RAW:*
    - Access to the network level directly (for example: IP level).
  - *SOCK_DGRAM:*
    - datagram mode (UDP).
- **protocol:** Specifies a particular protocol to be used with the socket. EX: 6 : TCP ; 17 UDP .

**Example:**

Listing 3.8: Example

```
1 int sockfd;
2 sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

The return value is -1 for Failure, otherwise a value identifying the descriptor.

### 3.4.2 Bind

- Give the address of the socket on the server.
- This function requests the system to assign a local address to a socket.
- In TCP / IP case: this function associated with the socket a local address and a port number.
- We must use this function to:
  - Receive connections (in the connected mode), or
  - Receive messages (in the disconnected mode).

**Syntax:**

Listing 3.9: Syntax of Bind

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 int bind ( int sid, struct sockaddr * addrptr , socklen_t len)
```

Where:

- **sid:** is the socket id.
- **addrptr:** is a pointer to the address family dependent address structure
- **len:** is the size of **addrPtr*.

When using internet sockets, the second parameter of bind (of type sockaddr in *) must be cast to (sockaddr *).
Return value is 0 if OK, or else -1, then *errno* gives the reason. The main reasons are:

- EBADF: sid is not a valid descriptor.
- EINVAL: socket has an address (bind already done).
- EACCES: The user has not enough privileges.
- ENOTSOCK: sid is not a socket descriptor (socket should be called first).

**Example:**

Listing 3.10: Example of using Bind

```
int st;
int portserveur;
int s,soc;
struct sockaddr_in serveur;
printf("Entrer the port number of server : ");
scanf("%d",&portserveur);
printf("Appel fonction socket\n");
soc = socket (AF_INET,SOCK_STREAM,6);
if (soc==-1)
{
        printf("invalid Socket");
        return (-1);
}
serveur.sin_family = AF_INET;
serveur.sin_port = htons(portserveur);
serveur.sin_addr.s_addr = inet_addr("127.0.0.1");
st = bind(soc,(const struct sockaddr *)&serveur,sizeof(serveur));
```

Where *inet_addr* is the conversion from pointed decimal format to hexadecimal format.

### 3.4.3 Listen

It specifies the maximum number of connection.
**Syntax:**

Listing 3.11: Syntax of listen

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

int listen (int sid, int size);
```

Where:

- **Sid:** socket
- **Size:** maximum number of connection

The return values are 0 if OK or else -1, then *errno* gives the reason. Main reasons:

- EADDRINUSE: The port number is already used by another socket.
- EBADF: socket is not valid (must call first socket and bind).
- ENOTSOCK: Similar to the previous case.
- EOPNOTSUPP: the handle descriptor is not intended to listen (connected or online mode).

**Example:**

Listing 3.12: Example of listen

```
1 if (listen(socket,10) == SOCKET_ERROR)
2 {
3         // Error treatment
4 }
```

### 3.4.4 connect

Connect to a server by providing address and port number of the server.
**Syntax:**

Listing 3.13: Syntax of connect

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 int connect(int sid,struct sockaddr *addrPtr,socklen_t addrlen len)
```

The return values are 0 if OK or else -1, then *errno* gives the reason. Main reasons:

- EBADF: invalid socket.
- EFAULT: The socket structure Address is outside the user's address space.
- ENOTSOCK: you must call the socket function before.
- EISCONN: already connected socket.
- ECONNREFUSED: the server is not listening.
- ETIMEDOUT: Timeout.
- ENETUNREACH: Network problem.

**Example:**

Listing 3.14: Example of connect

```
int s,st;
struct sockaddr_in server;
int portserver;
printf("Entry the port number of the server:\n");
scanf("%d",&portserver);
printf("call the socket functiont\n");
s = socket (AF_INET,SOCK_STREAM,6);
if(s==-1)
{
        printf(" invalide socket; ");
        return (-1);
}
printf ("connection request");
server.sin_family = AF_INET;
server.sin_port = htons(portserver);
st = connect(s,(const struct sockaddr *)&server,sizeof(server));
```

### 3.4.5 accept

- Launch the connection with a specific client, allows connection by accepting a call.
- Function used by the server to accept connection requests (function connect).
- This function creates a new socket corresponding to the connection and does not affect or change the socket that listens for connection requests.
- Typically should be called fork after accept to continue the dialogue with the client and to listen for new connection requests.
  - Fork():
    - create a child process
    - creates a new process by duplicating the calling process
    - After an accept, a child process is forked off to handle the connection.
    - Syntax: **int fork()**

**Syntax:**

Listing 3.15: Syntax of accept

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

Where:

- **S:** socket to listen for connection requests (bind and listen), the previously opened socket (local socket)
- **Addr:** client address

- **Addrlen:** size of the client address. This parameter must be initialize the size of the sockaddr.
- **socklen_t:**
  - Socket addresses of type length.
  - It is declared as integer.

**Return Values:**

- Returns the socketId and address of client connecting to socket
- If error: return -1. Then errno gives the reason. Main reasons:
  - EAGAIN or EWOULDBLOCK for calls non-blocking.
  - EBADF: s is invalid.
  - ENOTSOCK: s is not a socket.
  - EOPNOTSUPP : connected mode is not supported by s.
  - EINVAL: listen not yet known.
  - EPERM: blockage by a firewall.

**Example:**

Listing 3.16: Syntax of accept

```
int st;
int portserveur;
socklen_t lgclient;
int s,soc;
struct sockaddr_in server,client;
printf("TCP Server Programme \n");
printf("Enter the server port number: ");
scanf("%d",&portserver);
printf("call socket functino\n");
soc = socket (AF_INET,SOCK_STREAM,6);
if (soc==-1) {
        printf("invalid socket ");
        return (-1);}
server.sin_family = AF_INET;
server.sin_port = htons(portserveur);
server.sin_addr.s_addr = inet_addr("127.0.0.1");
st = bind(soc,(const struct sockaddr *)&server,sizeof(server));
if (st!=0) {
        printf("invalide bind; st=%d\n",st);
        exit(-1);}
printf("Waiting for connection requests on port %d\n",portserveur);
st = listen(soc,5);
if (st!=0) {
        printf(" invalid listen; ");
        return (-1);
}
s = accept(soc,(struct sockaddr *)&client,&lgclient);
```

### 3.4.6 Send/Sendto

- **Send:** is used to send a message in connected mode.
- **Sendto:** is used to send a message in connected or not connected mode.

**Syntax:**

Listing 3.17: Syntax of Send/Sendto

```
1 int send(int s, const void *msg, size_t len, int flags);
2 int sendto(int s, const void *msg, size_t len, int flags,
3 const struct sockaddr *to, socklen_t tolen);
```

Where:

- **S:** socket correspond with the communication.
- **Msg:** data to be sent (in general character array).
- **Len:** size in bytes of data to be transmitted.
- **Flags:** parameters of the transmission (urgent data, force transmission, synchronize, not segmented, etc ...) Use 0 for a normal transmission.
- **to:** destination address.
- **Tolen:** size of the destination address.

**Return Values:**
Send/Sendto returns the number of bytes sent or -1 if failure.
**Example:**

Listing 3.18: Example of Send/Sendto

```
1 int main()
2 {
3 int st;
4 int s;
5 struct sockaddr_in serveur;
6 char * adrserveur;
7 struct sockaddr name;
8 int lgname;
9 char * mes;
10 s = socket (AF_INET, SOCK_STREAM, 6)
11 ...connect...
12 printf("Entrer un message a envoyer au serveur ou fin
13 pour terminer :\n");
14 scanf("%s",mes);
15 st = send(s,mes,strlen(mes),0);
16 }
```

### 3.4.7 recv/recvfrom

*recv* is used to receive a message in connected mode, while *recvfrom* is used to receive a message in connected or not connected mode.

**Syntax:**

Listing 3.19: Syntax of recv

```
1 int recv (int s, void * buf, size_t len, int flags);
```

Listing 3.20: Syntax of recvfrom

```
1 int recvfrom (int s, void * buf, size_t len, int flags,
2 struct sockaddr * from, socklen_t * fromlen);
```

- **S:** socket correspond with the communication.
- **Buf:** received data (in general character array)
- **Len:** size in bytes of data received. It must be initialized to the size of the reception area to tell the system the size expected by the calling program for that area.
- **Flags:** parameters of the transmission . Use 0 for a normal transmission.
- **From:** address of the remote machine. This parameter is used to retrieve the address of the sender of the message.
- **Tolen:** size of from.

**Return Values:**
*recv/recvfrom* returns the number of bytes received or -1 if failure.

### 3.4.8 shutdown

End reading or writing.

**Syntax:**

Listing 3.21: Syntax of shutdown

```
1 int shutdown ( int sid , int how)
```

Disables sending (how=1 or how=2) or receiving (how=0 or how=2). Returns -1 on failure.

# Exercises

## Exercise 7

Write an echo program between a client and a server (in TCP and UDP protocol). This program allows:

- If the client requests a connection, the server response with the message Welcome.
- If the client sends a message, when the message is received by the server, the server replies by the same message. While the message! =end.

### Answer of exercise 7

**TCP Client:**

Listing 3.22: TCP Client

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <arpa/inet.h>
7 #include <unistd.h>
8 #include <string.h>
9 #define MAXLG 132 // maximum length of messages
10 int main()
11 {
12 int st,af,type,prot;
13 int end = 0;
14 int portserver;
15 int s;
16 struct sockaddr_in server;
17 char * adrserver;
18 struct sockaddr name;
19 // int lgname;
20 char * mes;
21 adrserveur = (char *) malloc(32);
22 mes = (char *) malloc(256);
23 printf("Client TCP Program\n");
24 printf("Enter the server address in the form of a.b.c.d :\n");
25 do
26 {
27         scanf("%s",adrserver);
28         if (inet_aton(adrserver, &server.sin_addr)!=0)
29                 end=2;
30         else
31                 printf("invalid Adresse, restart: ");
32 } while (end==0);
33 if (end==2)
34 {
35         printf("Enter the server port number :\n");
36         scanf("%d",&portserver);
37         printf("call the socket function\n");
38         s = socket (AF_INET, SOCK_STREAM, 6);
39         if (s==-1)
40         {
```

```
41                  Printf("invalide socket;");
42                  Return(-1);
43          }
44          printf ("connection request");
45          server.sin_family = AF_INET;
46          server.sin_port = htons(portserver);
47          st = connect(s,(const struct sockaddr *)&serveur,sizeof(serveur
                ));
48          if (st!=0)
49          {
50                  printf("Connection failed");
51                  return (-1);
52          }
53          printf("Connection set; wait welcome message \n");
54          st = recv(s, &mes[0],256,0);
55          if (st > 0)
56          {
57                  mes[st] = 0;
58                  printf("Message receive : %s\n", mes);
59          }
60          else if(st<0)
61          {
62                  printf ("Error on the recv function");
63                  return(-1);
64          }
65          // the connection is set, dialogue phase
66          do
67          {
68                  printf("Enter a message to be send to server or end to
                        finish:\n");
69                  scanf("%s",mes);
70                  st = send(s,mes,strlen(mes),0);
71                  if (st!=strlen(mes))
72                          printf("the message not completely sent or
                                connection failed");
73                  else
74                  {
75                          printf("Message successfully sent; wait the
                                server response \n");
76                          st = recv(s,&mes[0],256,0);
77                          if (st > 0)
78                          {
79                                  mes[st] = 0;
80                                  printf("server Response: %s\n",mes);
81                          }
82                  else
83                          printf("Problem in the server response
                                reception; %n",st);
84                  }
85          }while (strcmp(mes,"end")!=0);
86          }
87 }
```

**TCP Server:**

Listing 3.23: TCP Server

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
int main()
{ int st;
int portserver;
socklen_t lgclient;
int s,soc;
struct sockaddr_in server,client;
char * mes;
mes = (char *)malloc(256);
printf("TCP server program\n");
printf("Enter the server port number: ");
scanf("%d",&portserver);
printf("call the socket function \n");
soc = socket (AF_INET,SOCK_STREAM,6); // creation du socket
if (soc==-1)
{
        printf(" invalid socket");
        return (-1);
}
// Indication to system the server address and port number
serveur.sin_family = AF_INET;
serveur.sin_port = htons(portserveur);
serveur.sin_addr.s_addr = inet_addr("127.0.0.1");
st = bind(soc,(const struct sockaddr *)&serveur,sizeof(serveur));
if (st!=0)
{
        printf(" Invalid bind: st=%d\n",st);
        exit(-1);
}
printf(" wait a connection request on theport %d\n",portserveur);
st = listen(soc,5);
if (st!=0)
{
        printf("invalid listen; ");
        return (-1);
}
s = accept(soc,(struct sockaddr *)&client,&lgclient);
if (s==-1)
{
        printf("invalid accept; ");
        return -1;
}
printf("Connection accepted; client %s\n",inet_ntoa(client.sin_addr));
// the connection is set
mes = "Welcome";
st = send(s,mes,strlen(mes),0);
```

```
55 if (st==-1)
56 {
57         printf("Error on sending the message to the client ");
58         return -1;
59 }
60 char rep[128];
61 do {
62         printf("wait client message \n");
63         st = recv(s, &rep[0],256,0);
64         if (st > 0)
65         {
66                 rep[st] = 0;
67                 printf("characters number received %d",st);
68                 printf(" - Message received from the client : %s\n",
                       rep);
69                 st = send(s,rep,strlen(rep),0); // emission du message
                       en echo
70         }
71         else
72         {
73                 printf("Retour recv %n \n");
74                 st = sleep(3);
75         }
76 } while (strcmp("end",rep)!=0);
77 close(s);
78 close(soc);
79 }
```

**UDP Client:**

Listing 3.24: UDP Client

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <arpa/inet.h>
7 #include <unistd.h>
8 #include <string.h>
9 int main()
10 {
11 int st,af,type,prot;
12 int end = 0;
13 int portserver;
14 int s;
15 struct sockaddr_in server;
16 socklen_t lgserver;
17 char * adrserver;
18 struct sockaddr name;
19 int lgname;
20 char * mes;
21 adrserver = (char *) malloc(32);
22 mes = (char *) malloc(256);
23 printf("UDP Client UDP\n");
24 printf("Enter the server address in the form of a.b.c.d :\n");
25 do
26 {
```

```
27          scanf("%s",adrserver);
28          if (inet_aton(adrserver, &server.sin_addr)!=0)
29                  end=2;
30          else
31                  printf("invalid Adresse, restart: ");
32 } while (end==0);
33 if (end==2)
34 {
35          printf("Enter the server port number:\n");
36          scanf("%d",&portserver);
37          printf("call the socket function\n");
38          s = socket (AF_INET,SOCK_DGRAM,17);
39          if (s==-1)
40          {
41                  printf("invalid socket ; ");
42                  return (-1);
43          }
44          server.sin_family = AF_INET;
45          server.sin_port = htons(portserveur);
46          do {
47                  printf("Enter a message to be send to server or end to
                        finish:\n");
48                  scanf("%s",mes);
49                  st = sendto(s, mes, strlen(mes),0,(struct sockaddr *)&
                        serveur,sizeof(serveur));
50                  if (st!=strlen(mes))
51                          printf("the message not sent completely or
                                connection failed");
52                  else
53                  {
54                          printf("Message successfully sent; wait the
                                server response \n");
55                          lgserveur = sizeof(serveur);
56                          st = recvfrom(s,&mes[0],256,0,(sockaddr*)&
                                serveur,&lgserveur);
57                          if (st > 0)
58                          {
59                                  mes[st] = 0;
60                                  printf("server Response: %s\n",mes);
61                          }
62                          else
63                                  printf("Problem in the server response
                                        reception; %n",st);
64                  }
65          }while (strcmp(mes,"end")!=0);
66          }
67 }
```

**UDP Server:**

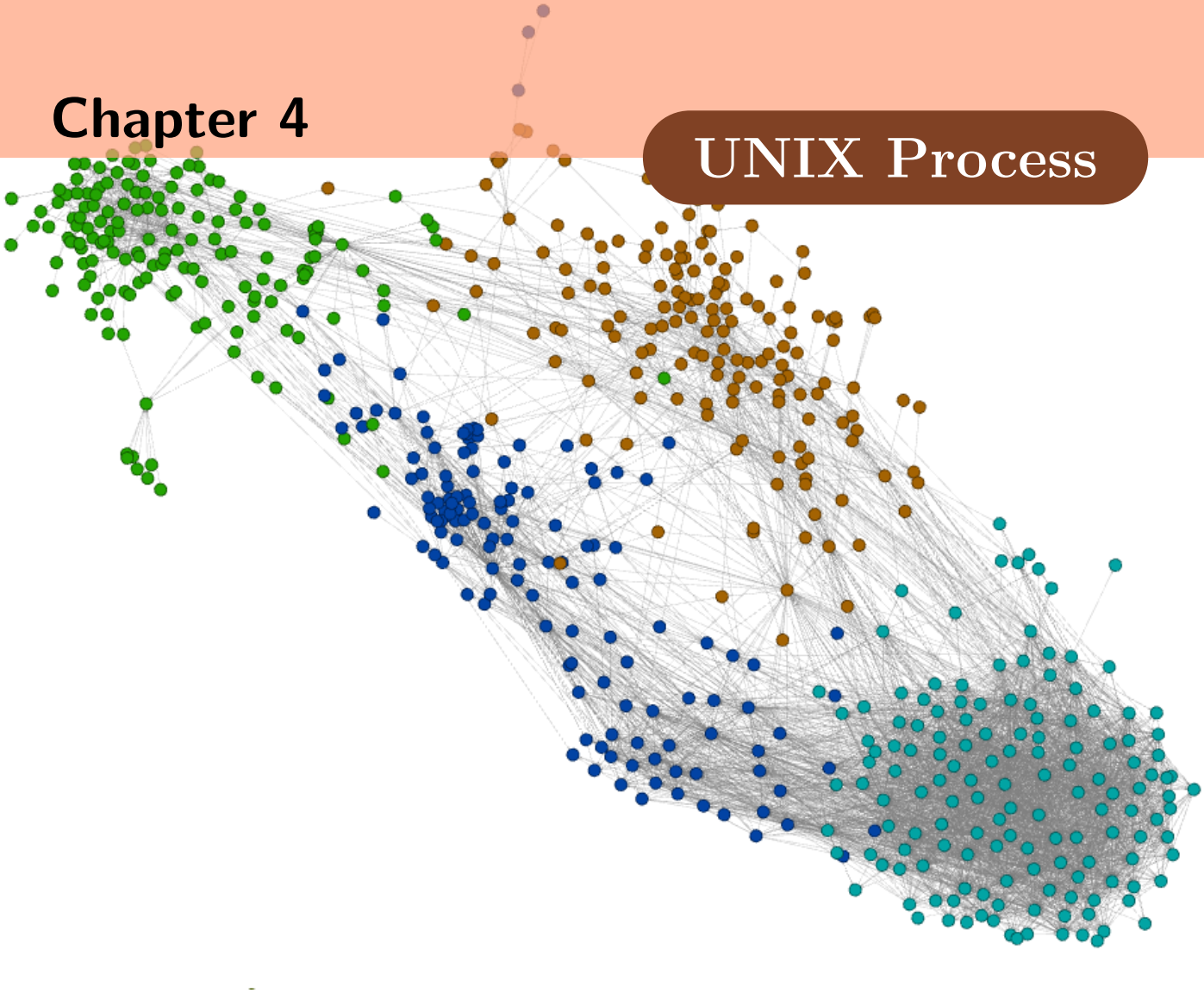
Listing 3.25: UDP Server

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <arpa/inet.h>
```

```c
#include <unistd.h>
#include <string.h>
#include <errno.h>
int main()
{
int st;
int portserver;
socklen_t lgclient;
int soc;
struct sockaddr_in server,client;
char * mes;
mes = (char *)malloc(256);
printf("UDP Server Program\n");
printf("Enter the server port number:: ");
scanf("%d",&portserver);
printf("call the socket function\n");
soc = socket (AF_INET,SOCK_DGRAM,17); // socket creation
if (soc==-1)
{
        printf("invalid socket");
        return (-1);
}
// Indication to system the server address and port number
server.sin_family = AF_INET;
server.sin_port = htons(portserveur);
server.sin_addr.s_addr = inet_addr("127.0.0.1");
st = bind(soc,(const struct sockaddr *)&server,sizeof(server));
if (st!=0)
{
        printf("invalid bind; st=%d\n",st);
        exit(-1);
}
printf("wait a messages on the port %d\n",portserver);
char rep[128];
do {
        printf("wait client message \n");
        st = recvfrom(soc,&rep[0],256,0,(sockaddr*)&client,&lgclient);
        if (st > 0)
        {
                rep[st] = 0;
                printf("characters number received %d",st);
                printf(" - Message received from the client: %s\n",rep)
                    ;
                st = sendto(soc,rep,strlen(rep),0,(struct sockaddr *)&
                    client,sizeof(client));
                //echo message emission
        }
        else
        {
                printf("Retour recv %n \n");
                st = sleep(3);
        }
} while (strcmp("fin",rep)!=0);
close(soc);
}
```

# Chapter 4

# UNIX Process



## 4.1 Introduction

A process is an instance of running a program. For example, three people are running a program at the same time; there are three processes there, not just one.

A process has two memory areas:

- Program Zone (executable machine code)
- Data Zone contains the data and the pile (for local variables and management functions).

UNIX identifies every process by a *Process Identification Number (pid)* which is assigned when the process is initiated.

When we want to perform an operation on a process, we usually refer to it by its pid.

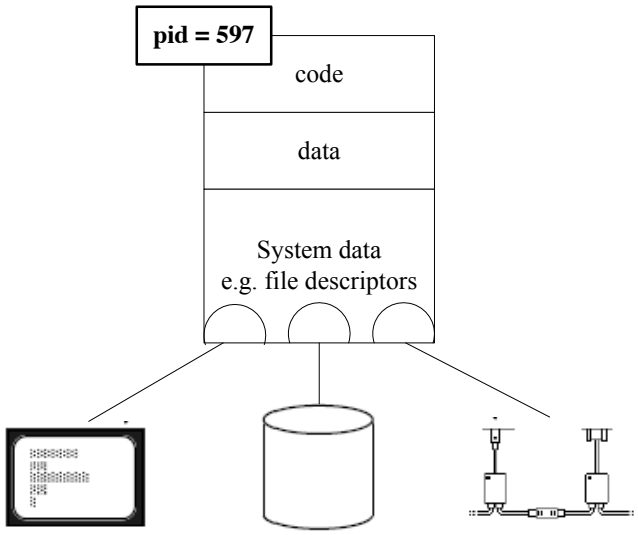



Figure 4.1: Process Identification Number (pid)

The **ps -x** command will list all your currently-running jobs. For example :

Listing 4.1: Union

```
1 PID TT STAT TIME COMMAND
2 6799 co IW 0:01 -csh[rich] (csh)
3 6823 co IW 0:00 /bin/sh /usr/bin/X11/startx
4 6829 co IW 0:00 xinit /usr/lib/X11/xinit/xinitrc --
5 6830 co S  0:12 X :0
```

The meaning of the column titles is as follows:

- **PID:** process identification number.
- **TT:** controlling terminal of the process.
- **STAT:** state of the job.
- **TIME:** amount of CPU time the process has acquired so far.
- **COMMAND:** name of the command that issued the process.

## 4.2 Parent PID (PPID)

Each process has a parent process that launch it. The identification number of the parent process is denoted by PPID (parent PID). A process of course has only one *parent*, but can start running on several others processes, named *child process*. We are thus dealing with a process tree.

## 4.3 Process Handling (in C language)

### 4.3.1 Fork ()

- Duplication of the process.

- Old process called the parent.
- new process called the child.
- Copies complete process state:
  program data + system data, including file descriptors

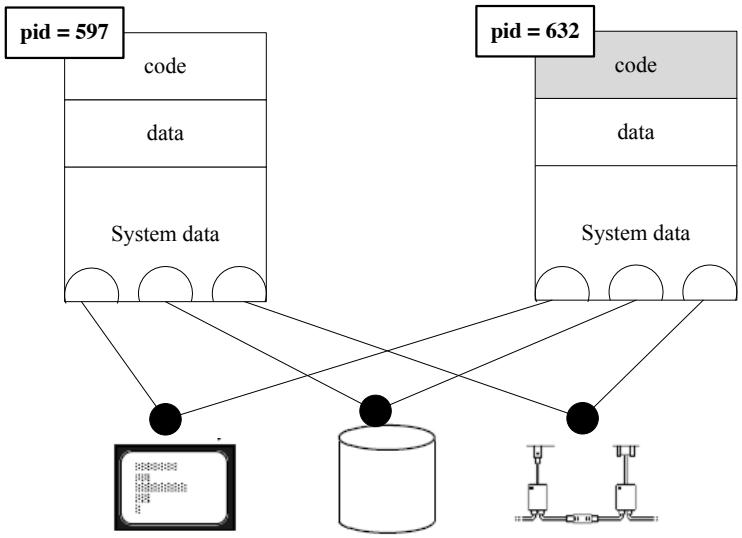



Figure 4.2: fork() scheme

**Syntax:**

Listing 4.2: Union

```
1 #include <unistd.h>
2 int fork();
```

fork() returns an integer. It has 2 cases:

1. If successful:
   - 0 in the child.
   - pid of the parent.
2. If failure:
   - -1 in the parent.
   - The child is not created.

**Example:**

### 4.3.2 getpid () and getppid()

- getpid () returns the PID of the current process → process ID.
- getppid () returns the PID of the parent process → parent process ID.

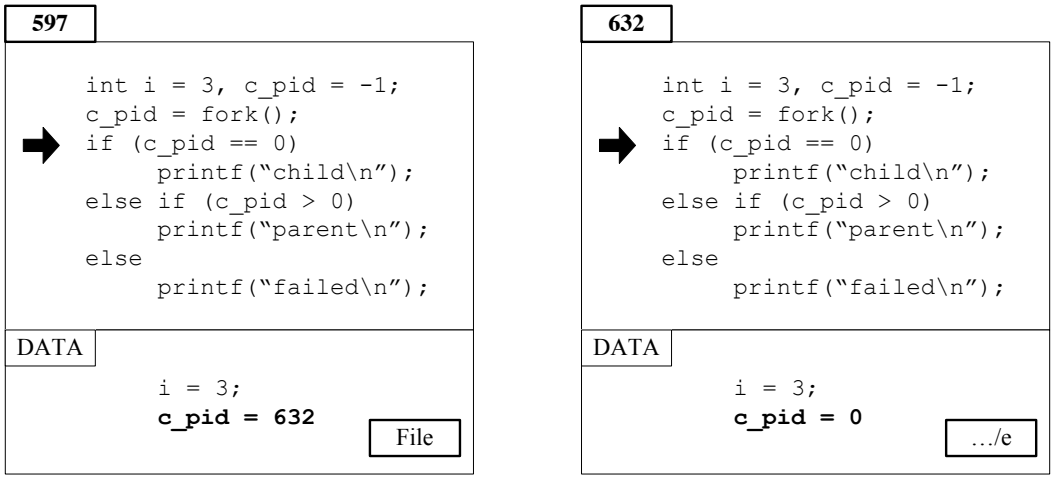



Figure 4.3: fork() example

### 4.3.3 Exit (int status)

Terminates the process by returning status to the parent.

Listing 4.3: Exit (int status)

```
1 #include <stdlib.h>
2 void exit (int status);
```

The *status* parameter is an integer that indicates that an error has occurred. Leave it to zero to indicate a normal exit.

### 4.3.4 Sleep(int seconds)

Sleep seconds seconds.

Listing 4.4: Sleep(int seconds)

```
1 #include <unistd.h>
2 int sleep( int seconds );
```

The process is blocked during sleep calls the specified number of seconds, unless it receives a signal.
**Example:**

Listing 4.5: Sleep example

```
1 for(;;) {
2         n = read(tty_fd,buff,buff_len);
3         if ( n > 0 ) { /* do something */ }
4         n = read(net_fd,buff,buff_len);
5         if ( n > 0 ) { /* do something */ }
6         sleep(5);}
```

### 4.3.5 Wait

- Waiting for message or block until a child has died.
- Return pid of dead child, or -1 if all child are terminated.

Listing 4.6: Wait

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 int wait( int *st );
```

### 4.3.6 Kill

- Sends a signal to a process.

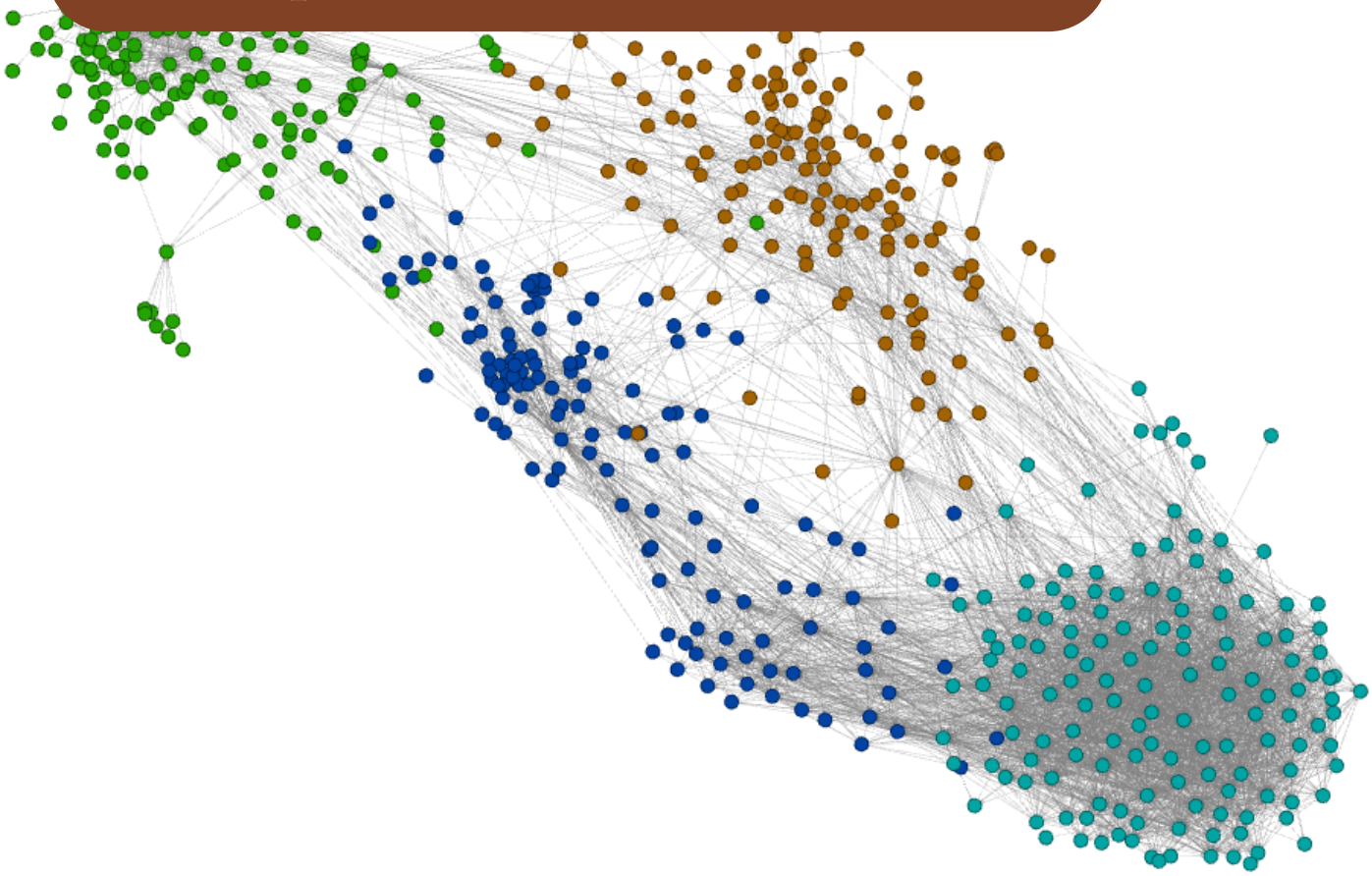
Listing 4.7: Kill

```
1 int kill(int pid, int sig);
```

- Sends the signal sig to process pid
- Target process → pid
- Return 0 in success case, otherwise -1 in failure case.

Chapter 5

# Inter-process Communication



## 5.1 Introduction

In computing, **Inter-process communication (IPC)** is a set of methods for the exchange of data among multiple threads in one or more processes. Processes may be running on one or more computers connected by a network.

UNIX provides several mechanisms for processes running on the same machine to share information.

- Pipes and Named Pipes.
- Shared Memory.
- Semaphores.
- Signals.
- Messages.
- UNIX Socket.

## 5.2 Pipes

The concept of pipes is illustrated in figure 5.1:

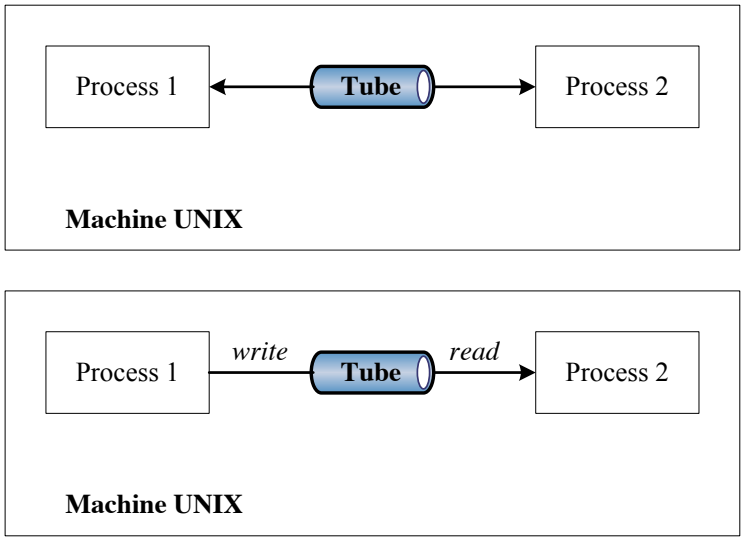



Figure 5.1: Concept of pipes

**Syntax:**

Listing 5.1: Syntax of pipes

```
1 #include <unistd.h>
2 int pipe(int p[2]);
```

If successful return 0 else return -1

- p[0]: readable file descriptor
- p[1]: writable file descriptor

**Emission:** use the write command:

Listing 5.2: Emission

```
1 int write (int fd, const void * , int );
```

*Where:*

- int fd: p[1].
- const void *: content of the message.
- int : size of the message.

**Reception:** use the read command:

Listing 5.3: Reception

```
1 int read (int fd, void *, int);
```

*Where:*

- int fd: p[0].
- void *: content of the message.
- int : size of the message.

We use pipe system call to link process to itself, as illustrated in figure 5.2.

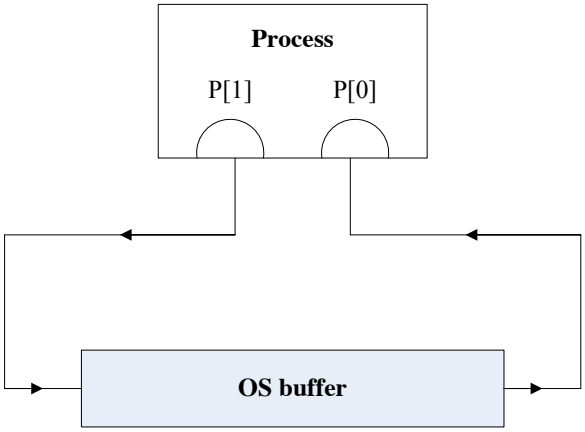



Figure 5.2: Pipe system call

We use fork file descriptors shared → both parent and child can read from p[0] and write to p[1] (figure 5.4).

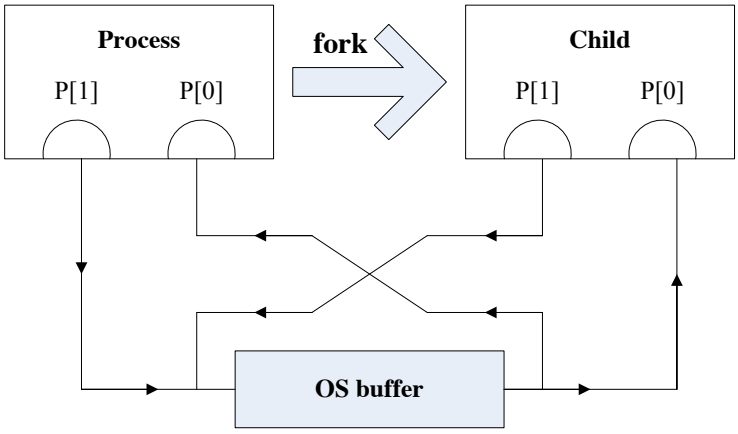



Figure 5.3: Pipe system call

## Exercise 8

Write the program code allow us to communicate between 2 processes parent and child, where the child created via the fork function, and the communication achieve using the pipe methodology.

- In the parent: we can entry a message and sent it to the child (while message != End).
- In the child: receive the message and display it on the screen.

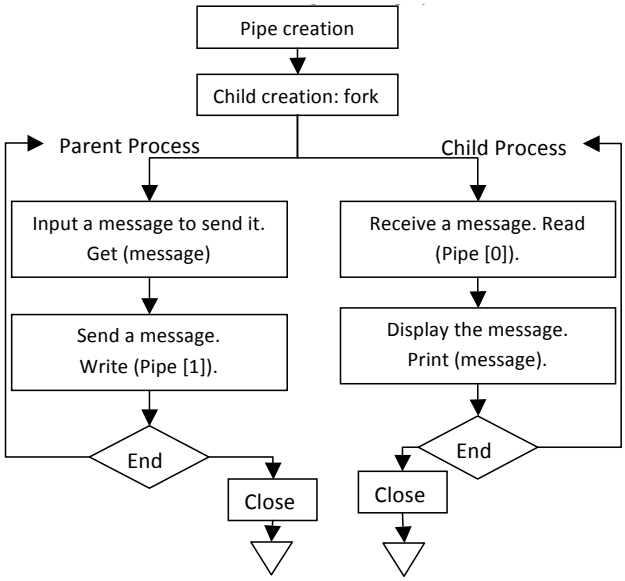



Figure 5.4: Flow Chart of exercise

### Answer of exercise 8

Listing 5.4: Simple communication via pipes

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
int main()
{
int p [2];
char mes[128];
int st;
printf("NSP : Example of fork usage and pipe \n");
printf("Create a pipe in the parent process \n");
if (pipe(p)>0)
{
        printf("Problem in the pipe creation procedure \n");
        return (1);
}
printf("call the fork function to create a child process \n");
switch (fork())
{
        case -1:
        perror("fork");
        return -1;
        case 0 : /* child process */
        do {
                st = read(p[0],mes,sizeof(mes));
                mes[st] = 0;
                if (st>0)
                        printf("received Message from parent %s\n",mes)
                            ;
```

```
28          } while (st>=0);
29          break;
30          default: // parent process
31          do {
32                  printf(" Enter a message that must be send to the child
                        process\n");
33                  gets(mes);
34                  printf("messsage Emission to the child process \n");
35                  write(p[1],mes,strlen(mes));
36          } while (mes!="end");
37 }
38 }
```

## Exercise 9

Write the program code allows us to communicate between 2 processes parent and child, where the child process created via the **fork** function and the communication achieve using the pipe methodology.

- In the parent: **(while message != End)**
  - Achieve communication pipe that allow the communication from Parent to Child.
  - Entry a message and sent it to the child (via pipe).
  - Wait for 2 seconds to receive the child response.
  - Receive the message from the client side and display it on the screen.
- In the child: **(while received message != )**
  - Receive the message from the parent process.
  - Achieve new communication pipe that allow the communication from Child to Parent.
  - Resend the same message to the parent.

### Answer of exercise 9

Listing 5.5: communicate between 2 processes parent and child

```
1 #include<stdio.h>
2 #include<string.h>
3 #include<unistd.h>
4 int main()
5 {
6 int pipe1[2],pipe2[2];
7 char mes[128];
8 int st;
9 printf("Creation two pipes 1 and 2 in the parent process\n");
10 if (pipe(pipe1)+pipe(pipe1)>0)
11 {
12          printf("Problem in the pipe creation procedure \n");
13          return(1);
14 }
15 printf("call the fork function to create a child process \n");
16
```

```
17 switch (fork())
18 {
19          case      1 :
20          perror("fork error");
21          return      1 ;
22          case 0:/* Child process */
23          do
24          {
25          st=read(pipe1[0],mes,sizeof(mes));
26          mes[st]=0;
27          if       (st>0)
28          {
29                   write(pipe2[1],mes,strlen(mes));
30          }
31          }while (st>=0);
32          break;
33          default:// Parent process
34          do
35          {
36                   printf(" Enter a message that must be send to the child
                         process\n");
37                   gets(mes);
38                   printf("Send message to the child process\n");
39                   write(pipe1[1],mes,strlen(mes));
40                   printf("wait for some seconds \n");
41                   sleep(2);
42                   st=read(pipe2[0],mes,sizeof(mes));
43                   mes[st]=0;
44                   if (st>0)
45                   printf("Child response:%s\n",mes);
46          }while(mes!="end");
47 }
48 }
```

## 5.3 Named Pipes (FIFOs)

- Problem pipes "simple":
  - We must know the result of the pipe function in order to communicate.
  - A process used between father and child process (fork).
- To communicate between two independent processes using named pipes:
  - Same principle as the pipe "simple".
  - Mechanism of creation is different.
- Named pipe (also known as a FIFO for its behavior) is an extension to the traditional pipe concept on Unix.
- Creation of the pipe: **mkfifo** *(make first in first out)*.

### 5.3.1 Syntax for the Creation of the Pipe

Listing 5.6: Syntax for the creation of the pipe

```
1 int mkfifo (const char * name, mode_t mode);
```

*Where:*

- **Name:** The name of the pipe (must be a valid filename)
- **Mode:** mode of creation; integer over 2 bytes representing the access permissions on the pipe. Use 0666 or O_CREAT.

This function returns 0 on success and -1 on failure.

### 5.3.2 Opening (Open)

Open a name pipe with flag indicate the access mode the named pipe (read/write).
**Syntax:**

Listing 5.7: Syntax for Opening a name pipe

```
1 int open (const char * pathname, int flags);
```

Possible values of flags:

- O_RDONLY: read only
- O_WRONLY: write only
- O_RDWR: read and write

This function returns a valid " descriptor file " on success or -1 on failure

### 5.3.3 Closure (close)

Close the access to the pipe.
**Syntax:**

Listing 5.8: Syntax for Closing the access to the pipe

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

Where *fd* is the file descriptor of the pipe to close.
This function returns 0 on success or -1 on failure.

### 5.3.4 Deletion (unlink)

Remove a named pipe name and the file that refers to it.
**Syntax:**

Listing 5.9: unlink

```
1 int unlink (const char * name);
```

Where *Name* is the name of the pipe
This function returns 0 on success or -1 on failure.

## Exercise 10

Write a program between 2 processes Pr1 and Pr2 (using a named pipe). This program allows:

1. The Pr1 allow creating a pipe named Demo, and read the message from this pipe and display it on the screen.
2. Pr2 allow to enter messages from the keyboard, write this message on the Demo pipe to send it to Pr1 (while the message! = end).

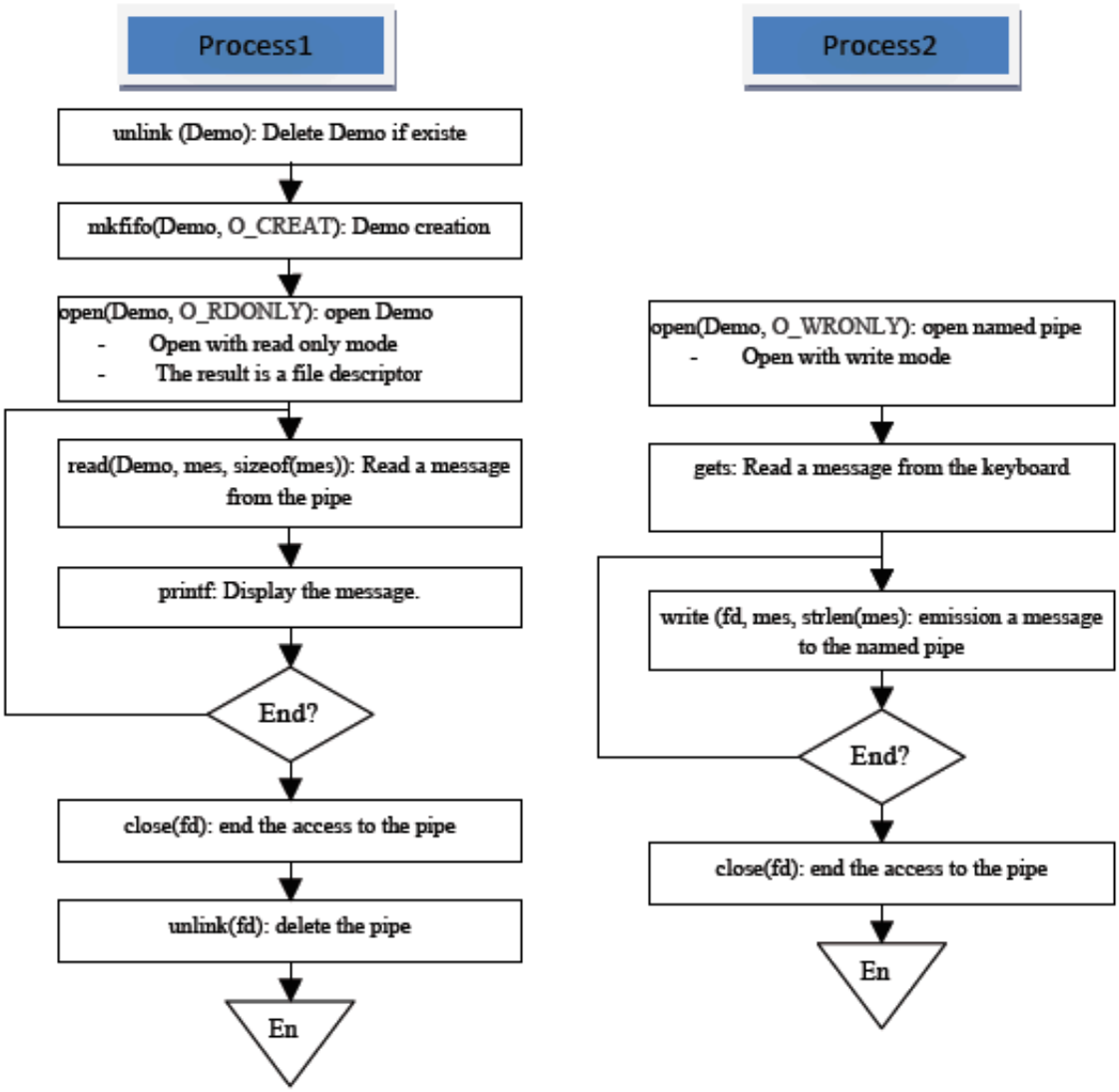



Figure 5.5: Two Processes

**Answer of exercise 10**

Listing 5.10: Simple communication via pipes

## 5.4 Shared Memory

- Multiple processes accessing the same memory zone
- The memory is accessed as part of the memory zone data of the process
- Advantage is *speed*; access is instant (no message transfer)

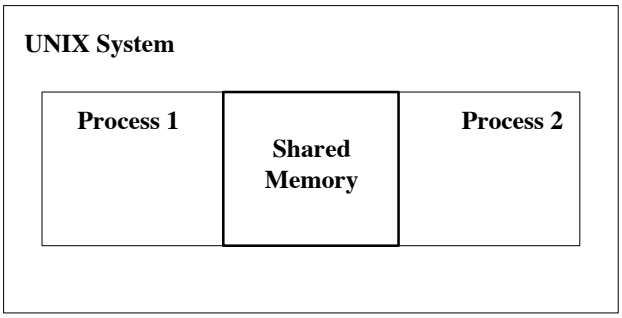



Figure 5.6: Shared Memory

*Operations* used in shared memory are:

- Creating a Zone.
- Attaching the process area.
- Access to data.
- Release of the area.

*Functions* used in shared memory are:

- **ftok:** identifier(key) generation function.
- **shmget:** creation function.
- **shmat:** attachment process in the area function.
- **shmctl:** control of the area (used for release).

### 5.4.1 Functions Declarations

**ftok**

Listing 5.11: ftok

```
1 key_t ftok (const char * name, int proj_id);
```

*Where:*

- **name:** pathname (which must refer to an existing, accessible file)
- **proj_id:** is the character upon which part of the key is formed.

**shmget**

Listing 5.12: shmget

```
1 int shmget (key_t key, int size, int shmflg);
```

**shmflg** takes the values:

- **IPC_CREAT** which has the effect of creating the zone if it is no exist
- **IPC_EXCL** returns an error if the zone already exists

**shmat**

Listing 5.13: shmat

```
1 void * shmat (int shmid, const void * shmaddr, int shmflg);
```

- **Shmaddr:** put NULL to let the system assign the address.
- **Shmflg:** SHM_RDONLY for read-only access. Set to 0 for read / write.

**shmctl**

Listing 5.14: shmctl

```
1 int shmctl (int shmid, int cmd, struct shmid_ds * buf);
```

**shmctl** returns 0 if Ok.

Values taken by **cmd**:

- **IPC_SET:** change access permissions (read / write).
- **IPC_RMID:** suppression of memory.
- **SHM_LOCK:** not allow switching the memory area.
- **SHM_UNLOCK:** allow switching.

Listing 5.15: struct shmid_ds

```
1 struct shmid_ds
2 {
3         struct ipc_perm shm_perm;    /* operation perms */
4         int shm_segsz;                        /* size of segment (
              bytes) */
5         time_t shm_atime;                    /* last attach time */
6         time_t shm_dtime;                        /* last detach
              time */
7         time_t shm_ctime;                        /* last change
              time */
8         unsigned short shm_cpid;                /* pid of creator */
9         unsigned short shm_lpid;                /* pid of last
              operator */
10        short shm_nattch;                        /* no. of current
              attaches */
11        ...
12 };
```

## 5.5 Signals

Signals are simply used to inform a process that an event has happened. There are about 32 predefined signals, each having a meaning. It is not possible to define new signals. Signals are sent either by the system itself (especially when the process attempts an illegal operation such as division by zero or access to a memory area not owned), or by another process belonging to the same user. Some signals, such as the number 9, will kill signals that are still in the process.

We can send a signal to another process by calling the primitive:

**shmat**

Listing 5.16: kill signal

```
1 int kill (int pid, int signum);
```

This command sends the signal number **signum** to the process **pid**.

Some signals are defined by all major UNIX systems.

| Value | Signal | Comment |
|---|---|---|
| 1 | HUP | End or death of controlling process |
| 8 | FPE | Floating point exception |
| 9 | KILL | Kill signal |
| 10 | USR1 | User-defined signal 1 |
| 11 | SEGV | Invalid memory reference |
| ... | | |

The complete list is defined on each system in the file include ¡*signal.h*¿.

To specify a C function to be called when a signal is received:

Listing 5.17: Signal syntax

```
1 void (* signal (int signum, (void * handler) (int)))) (int);
```

The handler argument is a C function that takes an integer argument and will be automatically called when the signal is received by the process.
**Example:**

Listing 5.18: Signal syntax

```
1 void treat _signal_usr1 (int signum)
2 {
3         printf ("received signal% d. \ n", signum);
4 }
```

This function is installed by calling:

Listing 5.19: Install the function

```
1 signal (SIGUSR1, treat _signal_usr1);
```

You can also specify that you want to ignore a signal type using the constant **SIG_IGN**:

Listing 5.20: Ignore a signal

```
1 signal (number_ignored_signal, SIG_IGN);
```

## 5.6 Socket Programming using .Net Frame Work & Visual Studio

We propose to write an echo server that will run from a DOS window using the C# Code. The server run on a port passed as a parameter. It just refers to the client's request that it sent him. Also, we propose to write a client for the previous server. It connects to the machine *hostName* on the port *port*, and then it sends ta request to the server and receives back an echo.

### 5.6.1 Server Echo

The server program is as follows:

Listing 5.21: Server Echo

```
1 using System.Net.Sockets;
2 using System;
3 using System.IO;
4 using System.Threading;
5
6 namespace ConsoleApplication1
7 {
8         class Program
9         {
10
11                 static void Main(string[] args)
12                 {
13                         int port = 0;
14
15                                 Console.WriteLine("Port number:");
16                                 port = int.Parse(Console.ReadLine());
17
18                         TcpListener listening = null;
19                         int nbClients = 0;
20                         try
21                         {
22                                 listening = new TcpListener(port);
23
24                                 listening.Start();
25
26                                 Console.Out.WriteLine("Echo server
                                       started on port: " + port);
27                                 Console.Out.WriteLine( listening.
                                       LocalEndpoint );
28
29                                 TcpClient ClientLink = null;
30                                 while (true)
31                                 {
32                                         ClientLink = listening.
                                               AcceptTcpClient();
33                                         nbClients++;
34                                         ClientEchoTreat CET = new
                                               ClientEchoTreat(ClientLink,
                                               nbClients);
35
```

```
36                                      ThreadStart THS = new
                                            ThreadStart((CET.Run));
37                                      Thread TH = new Thread(THS);
38                                      TH.Start();
39                                  }
40                          }
41                          catch (Exception ex)
42                          {
43                                  // on signale l'erreur
44                                  Console.WriteLine ("L'erreur suivante s
                                        'est produite : " + ex.Message, 3);
45                          }//catch
46                          // fin du service
47                          listening.Stop();
48                  }// fin main
49                  // affichage des erreurs
50          }
51
52          public class ClientEchoTreat
53          {
54                  private TcpClient ClientLink;
55                  private int numClient;
56                  private StreamReader IN;
57                  private StreamWriter OUT;
58
59                  public ClientEchoTreat(TcpClient ClientLink, int
                        numClient)
60                  {
61                          this.ClientLink = ClientLink;
62                          this.numClient = numClient;
63                  }
64
65                  public void Run()
66                  {
67
68                          Console.Out.WriteLine("Start of Client service"
                                + numClient);
69                          try
70                          {
71
72                                  IN = new StreamReader(ClientLink.
                                        GetStream());
73
74                                  OUT = new StreamWriter(ClientLink.
                                        GetStream());
75                                  OUT.AutoFlush = true;
76
77                                  string request = null;
78                                  string response = null;
79                                  while ((request = IN.ReadLine()) !=
                                        null)
80                                  {
81
82                                          Console.Out.WriteLine("Client "
                                                + numClient + " : " +
                                              request);
83
84                                          // response = "[" + request +
                                              "]";
```

```
85                                          response = Console.ReadLine();
86                                          OUT.WriteLine(response);
87
88
89                                          if (request.Trim().ToLower() ==
                                                "end") break;
90                                  }
91                                  ClientLink.Close();
92                          }
93                          catch (Exception e)
94                          {
95                                  Console.WriteLine("Error in the client
                                       connection (" + e + ")" );
96                          }
97                          Console.Out.WriteLine("End Client service " +
                               numClient);
98                  }
99          }
100 }
```

### 5.6.2 Client Echo

Listing 5.22: Client Echo

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Net.Sockets;
6  using System.IO;
7
8  namespace ConsoleApplication3
9  {
10         class Program
11         {
12                 static void Main(string[] args)
13                 {
14                         int port = 0;
15
16                         Console.WriteLine("Port number:");
17                         port = int.Parse(Console.ReadLine());
18
19                         string servername = "";
20
21                         Console.WriteLine("server name:");
22                         servername = Console.ReadLine();
23                         TcpClient client = null ;
24
25
26                         StreamReader IN = null;
27                         StreamWriter OUT = null;
28
29
30                         string request = null;
31                         string response = null;
32                         try
```

```
33                          {
34                                  client = new TcpClient(servername, port
                                        );
35                                  IN = new StreamReader(client.GetStream
                                        ());
36                                  OUT = new StreamWriter(client.GetStream
                                        ());
37                                  OUT.AutoFlush = true;
38                                  while (true)
39                                  {
40
41                                          Console.Write("resquest (end-->
                                                Stop:");
42                                          request = Console.ReadLine();
43                                          OUT.WriteLine(request);
44
45                                          response = IN.ReadLine();
46
47                                          Console.Out.WriteLine("Response
                                                : " + response);
48                                          if (request.ToString().Trim().
                                                ToLower() == "end") break;
49                                          }
50                                  }
51                                  catch { Console.WriteLine("errors"); }
52
53                                  }
54
55                          }
56                  }
```

## ‘So Calming.’

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

ISBN 978-80-85955-35-4

Malaysian LaTeX User Group • http://latex-my.blogspot.com
Cover Illustration by Dusan Bicanski • http://www.public-domain-image.com