Padua University

Computer Engineering Master Degree

Computer Networks

Raffaele Di Nardo Di Maio

Accademic Year 2019-2020

Contents

Chapter 1

OSI model

The *Open System Interconnection (OSI)* is the basic standardization of concepts related to networks (Figure 1.1). It was made by Internet *Standard Organization (ISO)*. Each computer, connected as a node in the network, needs to have all OSI functionalities.

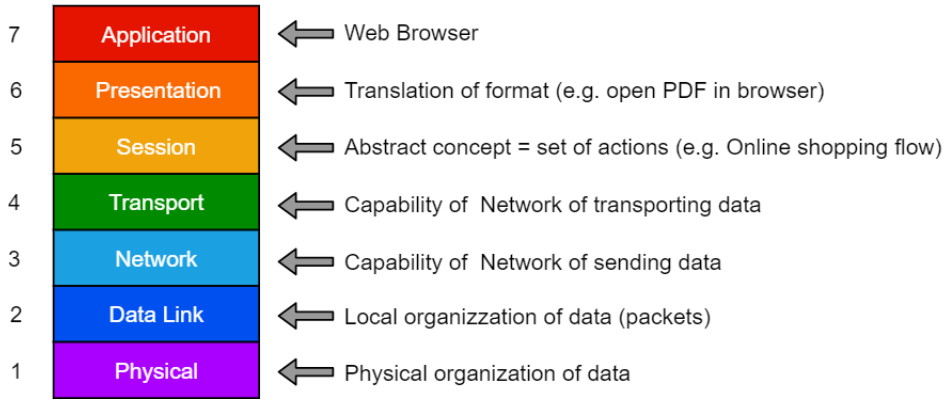
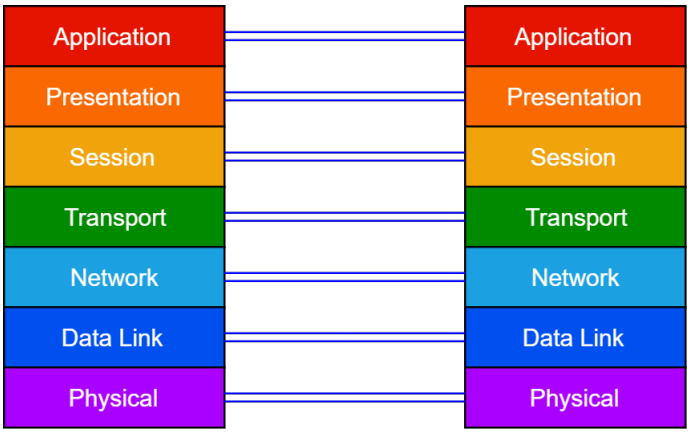


Figure 1.1: OSI model.

1.1 Logical communication



Layer 1 is the only one in which the real connection is also the logic connection. Each layer is a module (black-box) that implements functionality (see Section 1.4).

1.2 Control plane

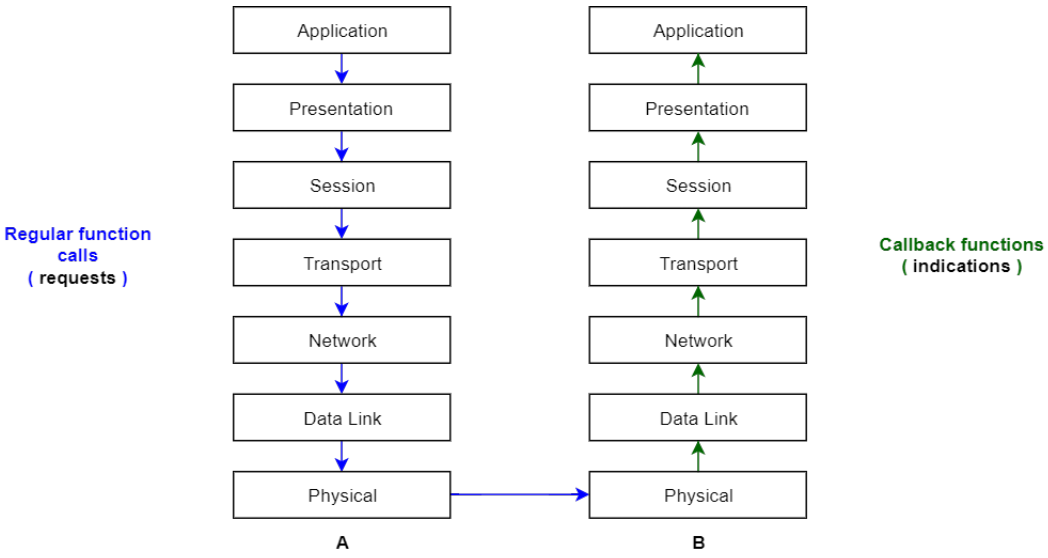


Figure 1.2: Request from A to B.

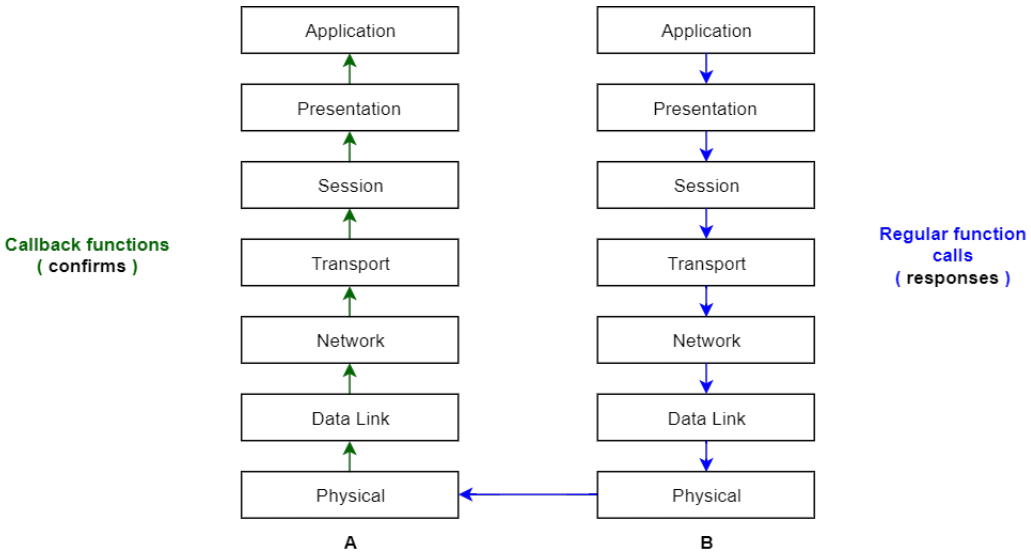


Figure 1.3: Response from B to A.

The control plane meaning comes from two words: "control" that is related to function activation and "plane", related to the geometry, because it's stacked in a sheet.
In OSI model, the *direct connection* exists only between:

- Upper and lower layers of the same device
- Physical layers of different devices

From Figure 1.2 and Figure 1.3 we have seen two main types of function calls:

- **Regular function calls**
  - library method invocations

  - system calls
  - HW enabled signals

- **Callback functions**
  the module of the upper layer is waken up by module of the lower layer.
  - OS signal handler
    it asks library to call a function when something happens (EVENT-BASED PROGRAMMING)
  - Interrupt handlers
  - Blocking function calls
    they start call but doesn't return if something doesn't happen

1.3 Data plane

Data plane defines which data are shared among the network. Calling a function, we need to pass parameters to them (*Data buffer*).
The PDU (Protocol Data Unit) of layer *i+1* becomes the SDU (Service Data Unit), or payload, of lower Layer *i*. Merging this payload, with the header of layer *i*, we obtain the PDU of layer i (Figure 1.4). This procedure is called **encpsulation** (Figure 1.5).

Figure 1.4: PDU and SDU structure.

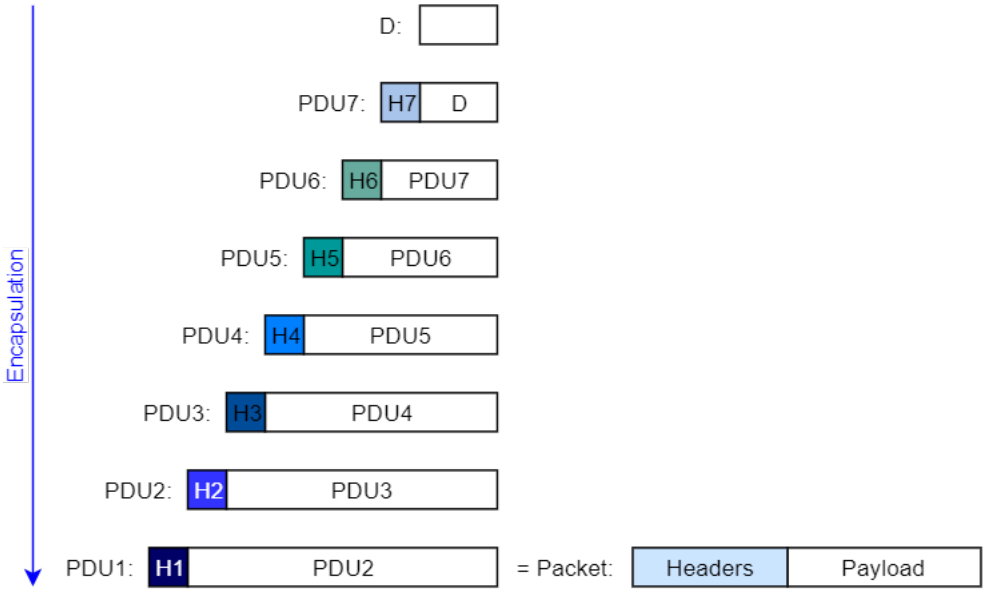


Figure 1.5: Encapsulation.

1.4 Onion model

The following image shows the layered structure of OS and computers and where OSI functionalities locations are highlighted.

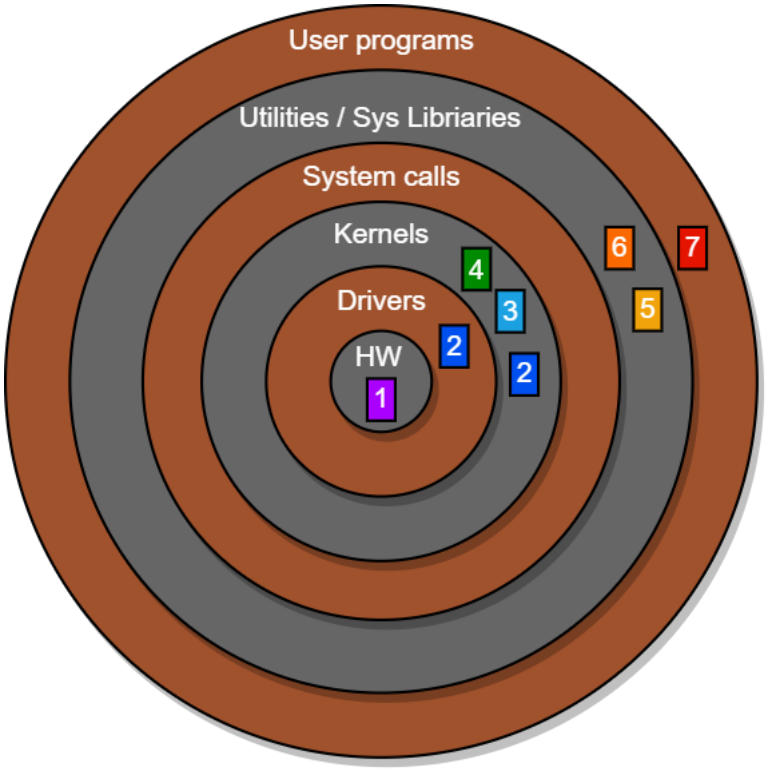
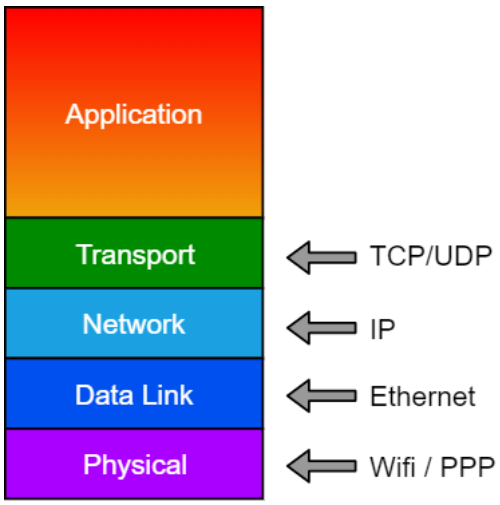


Figure 1.6: Onion model.

1.5 TCP/IP Architecture

The TCP/IP architecture is a reorganization of the previously mentioned OSI model (Figure 1.1) and it composes the main structure of the Internet Protocol.



1.6 Application paradigms

1.6.1 Client-Server

It's based on the presence of two main entities:

- **Client** = active entity
  it generates the request
- **Server** = passive entity
  it's waiting for client requests and when it receives it, it only replies to it.

The main characteristic of this paradigm is the **"immediate" response time**, that is the time between the arrival of the request by the client and the reply with the generate response.
To send the request, the client needs to know:

- server name
- how to reach it
- what data is required on server (trackable)

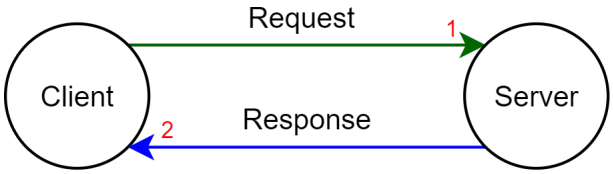


Figure 1.7: Client-Server architecture.

1.6.2 Peer-to-Peer (P2P)

Its diffusion started at first years of XXI century. It's used to share media. Each node in the network can be client (making requests) or server (replying to requests).
In Figure 1.8, $USER_1$ doesn't know which is the user in the network that shared the content. Hence, he sends the request for the content to a node in the network and this one can reply with two possible responses:

- **C**= content (media)
- **R**= reference to another node (that has the required content or knows which node has the content)

Each node can also forward the request to some other node and so it becomes the intermediary of the communication.

1.6.3 Publish/Subscribe/Notify

The subscriber subscribes to the dispatcher (notifier) a set of messages that wants to be notified. The notifier usually filters the messages that it receives and, when there are new messages that respect the subscription of the user, notify them to the user.
The messages comes *asynchronously* to the dispatcher. There is no *Polling* made periodically by the user (there isn't Busy Waiting). There are some applications, like Whatsapp, that work in this way but in the past, this app made by Facebook doesn't really work asynchronously. In fact there was a polling policy.

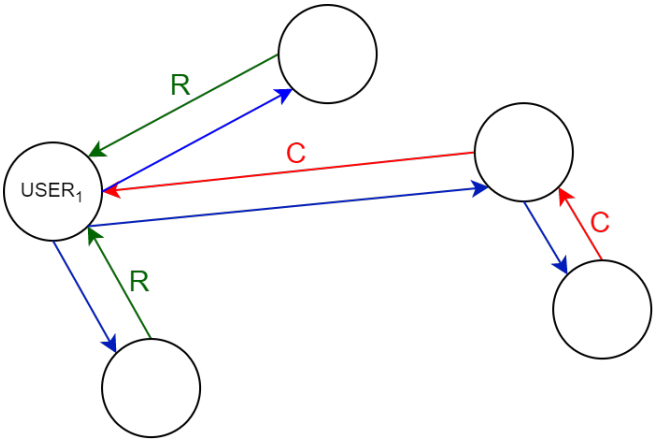


Figure 1.8: P2P architecture.

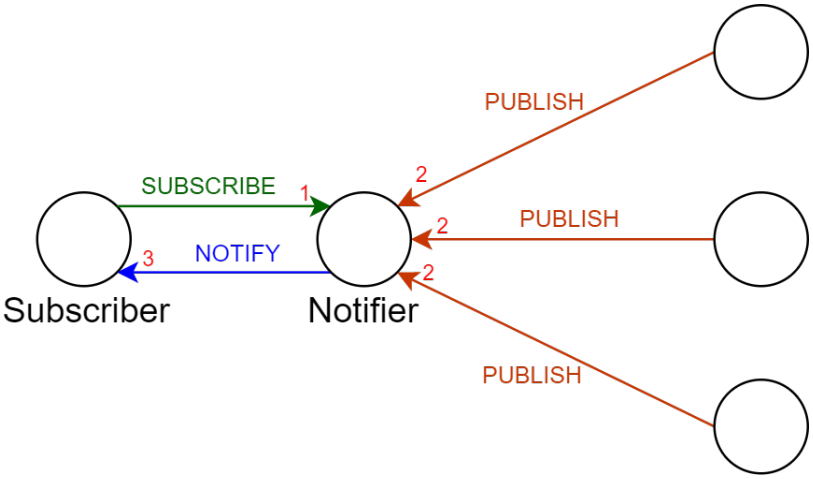


Figure 1.9: Publish/Subscribe/Notify architecture.

1.7 Types of packets

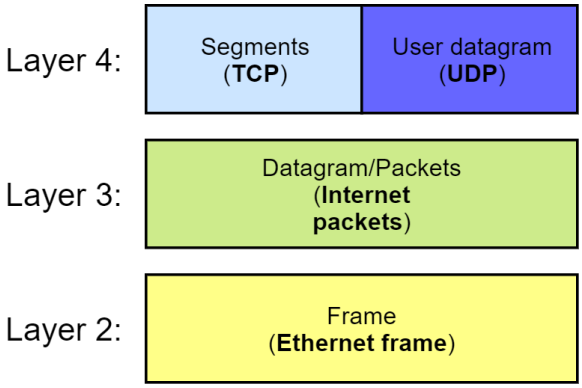


Figure 1.10: Standard names of packets.

TCP connection works at Layer 4 but at upper layers, it seems to work as a stream. In TCP connection, it is usually specified the port number, that is the upper layer protocol specification (Layer 5).

Chapter 2

C programming

The C is the most powerful language and also can be considered as the language nearest to Assembly language. Its power is the speed of execution and the easy interpretation of the memory.
C can be considered very important in Computer Networks because it doesn't hide the use of system calls. Other languages made the same thing, but hiding all the needs and evolution of Computer Network systems.

2.1 Organization of data

Data are stored in the memory in two possible ways, related to the order of bytes that compose it. There are two main ways, called Big Endian and Little Endian.

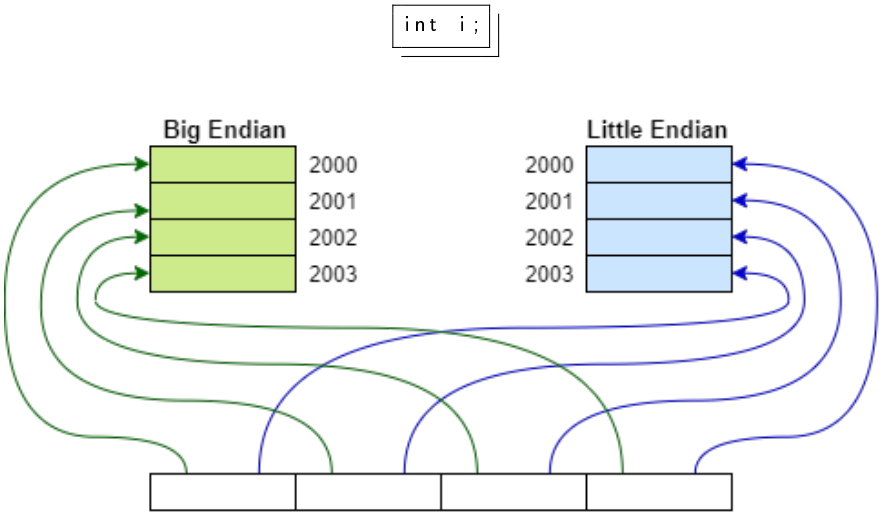


Figure 2.1: Little Endian and Big Endian.

The order of bytes in packets, sent through the network, is Big Endian.
The size of **int, float, char, ...** types depends on the architecture used. The max size of possible types depends on the architecture (E.g. in 64bits architecture, in one istruction, 8 bytes can be written and read in parallel).

signed	unsigned
int8_t	uint8_t
int16_t	uint16_t
int32_t	uint32_t
int64_t	uint64_t

Table 2.1: <stdint.h>

2.2 Struct organization of memory

The size of a structure depends on the order of fields and the architecture. This is caused by alignment that depends on the number of memory banks, number of bytes read in parallel. For example the size is 4 bytes for 32 bits architecture, composed by 4 banks (Figure 2.2). The Network Packet Representation is made by a stream of 4 Bytes packets as we're using 32 bits architecture.

```
struct example1          struct example2
{                        {
  char c;                  int x;
  int x;                   char c;
}                        }
```

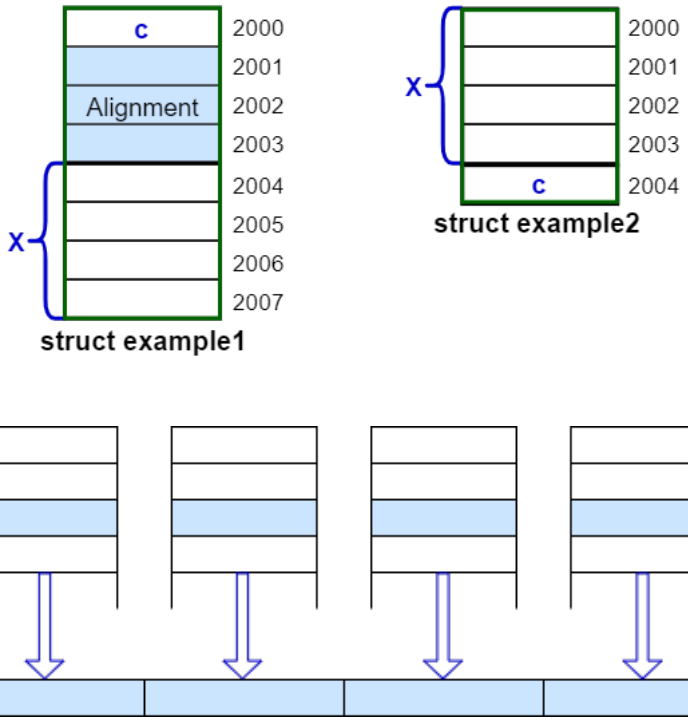


Figure 2.2: Parallel reading in one istruction in 32 bits architecture.

2.3 Structure of C program

The program stores the variable in different section (Figure 2.3):

- **Static area**
  where global variables and static library are stored, it's initialized immediately at the creation of the program. Inside this area, a variable doesn't need to be initialized by the programmer because it's done automatically at the creation of the program with all zeroes.
- **Stack**
  allocation of variables, return and parameters of functions
- **Heap**
  dinamic allocation

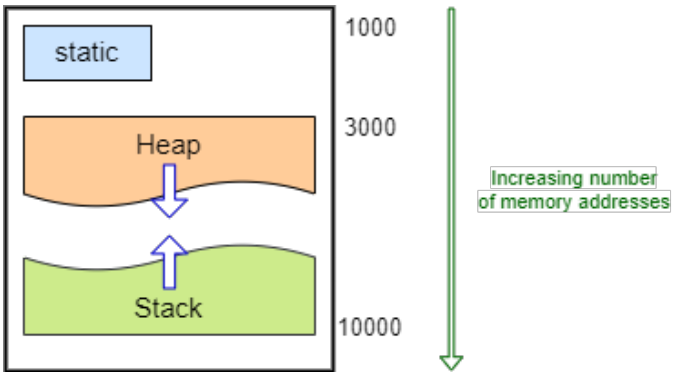


Figure 2.3: Structure of the program.

Chapter 3

Network in C

3.1 Application layer

We need IP protocol to use Internet. In this protocol, level 5 and 6 are hidden in Application Layer.
In this case, Application Layer needs to interact with Transport Layer, that is implemented in OS Kernel (Figure 3.1). Hence Application and Transport can talk each other with System Calls.

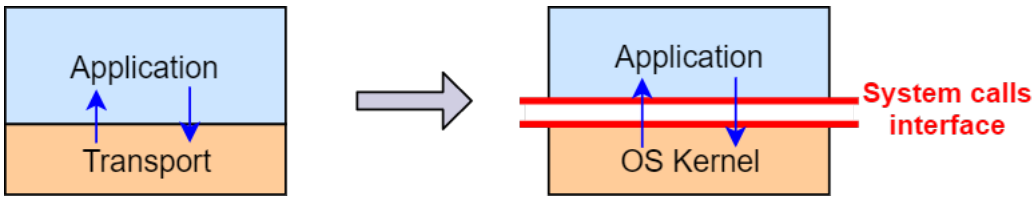


Figure 3.1: System calls interface.

3.2 socket()

Entry-point (system call) that allow us to use the network services. It also allows application layer to access to level 4 of IP protocol.

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

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

RETURN VALUE *File Descriptor (FD) of the socket*
-1 if some error occurs and errno is set appropriately
(You can check value of errno including <errno.h>).

domain = *Communication domain*
protocol family which will be used for communication.

AF_INET:	IPv4 Internet Protocol
AF_INET6:	IPv6 Internet Protocol
AF_PACKET:	Low level packet interface

type = *Communication semantics* (Figure 3.2)

SOCK_STREAM:	Provides sequenced, reliable, two-way, connection-based bytes stream. An OUT-OF-BAND data mechanism may be supported.
SOCK_DGRAM	Supports datagrams (connectionless, unreliable messages of a fixed maximum length).

protocol = *Particular protocol to be used within the socket*
Normally there is only a protocol for each socket type and protocol family (protocol=0), otherwise ID of the protocol you want to use

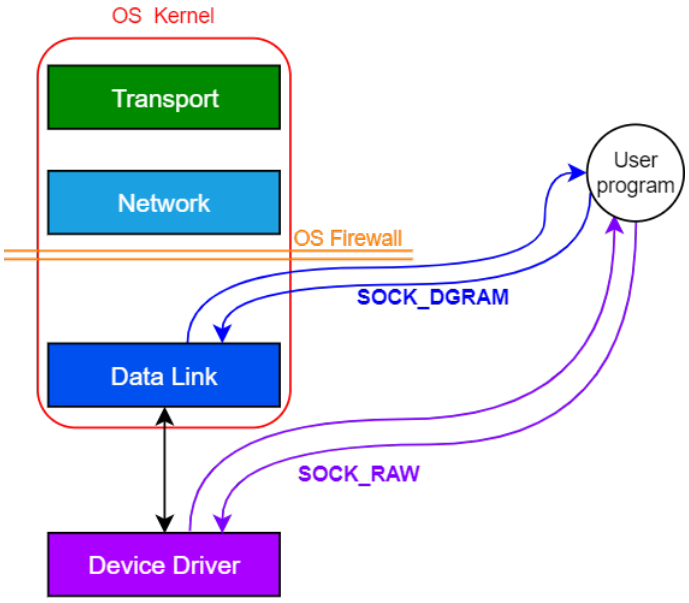


Figure 3.2: UNIX management.

3.3 TCP connection

In TCP connection, defined by type **SOCK_STREAM** as written in the Section 3.2, there is a client that connects to a server. It uses three primitives (related to File System primitives for management of files on disk) that do these logic actions:

1. start (open bytes stream)
2. add/remove bytes from stream
3. finish (clos bytes stream)

TCP is used transfering big files on the network and for example with HTTP, that supports parallel download and upload (FULL-DUPLEX). The length of the stream is defined only at closure of the stream.

3.3.1 Client

3.3.1.1 connect()

The client calls **connect()** function, after **socket()** function of Section 3.2. This function is a system call that client can use to define what is the remote terminal to which he wants to connect.

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

int connect(int sockfd, const struct sockaddr *addr,socklen_t addrlen);
```

RETURN VALUE *0* if connection succeds

-1 if some error occurs and errno is set appropriately

sockfd = *Socket File Descriptor* returned by socket().

addr = *Reference to struct sockaddr*

sockaddr is a general structure that defines the concept of address.

In practice it's a union of all the possible specific structures of each protocol.

This approach is used to leave the function written in a generic way.

addrlen = *Length of specific data structure used for sockaddr.*

In the following there is the description of struct **sockaddr_in**, that is the specific sockaddr structure implemented for family of protocls **AF_INET**:

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

struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};

/* Internet address. */
struct in_addr {
    uint32_t       s_addr;     /* address in network byte order */
};\\
```

The two addresses, needed to define a connection, are (see Figure 3.3):

- **IP address** (*sin_addr* in *sockaddr_in struct*)
  identifies a virtual interface in the network. It can be considered the entry-point for data arriving to the computer. *It's unique in the world.*
- **Port number** (*sin_port* in *sockaddr_in struct*)
  identifies to which application data are going to be sent. The port so must be open for that stream of data and it can be considered a service identifier. There are well known port numbers, related to standard services and others that are free to be used by the programmer for its applications (see Section A.2 to find which file contains well known port numbers). *It's unique in the system.*

As mentioned in Section 2.1, network data are organized as Big Endian, so in this case we need to insert the IP address according to this protocol. It can be done creating an array of char and analysing it as an int pointer* or with the follow function, that converts a string (E.g. "127.0.0.1") in the corresponding address:

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

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

If you want to obtain the IP address from the name of the host, using DNS, you need to use the following function that returns in h_addr_list the set of ip addresses related to that hostname, as arrays of characters:

```
#include <netdb.h>
extern int h_errno;

struct hostent *gethostbyname(const char *name);

struct hostent
{
    char  *h_name;            /* official name of host */
    char **h_aliases;         /* alias list */
    int    h_addrtype;        /* host address type */
    int    h_length;          /* length of address */
    char **h_addr_list;       /* list of addresses */
}
```

The port number is written according to Big Endian architecture, through the next function:

```
#include <arpa/inet.h>

uint16_t htons(uint16_t hostshort);
```

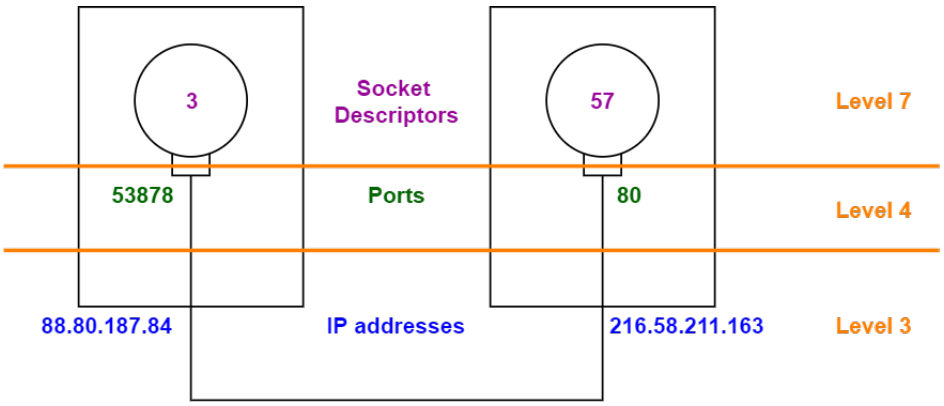


Figure 3.3: After successful connection.

3.3.1.2 write()

Application protocol uses a readable string, to excange readable information (as in HTTP). This tecnique is called simple protocol and commands, sent by the protocol, are standardized and readable strings.

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

The write buffer is usually a string but we don't consider the null value (\0 character), that determine the end of the string, in the evaluation of count (**strlen(buf)-1**). This convention is used because \0 can be part of characters stream.

RETURN VALUE *Number of bytes written* on success
-1 if some error occurs and errno is set appropriately

fd = *Socket File Descriptor* returned by socket().

buf = *Buffer of characters to write*

count = *Max number of bytes to write* in the file (stream).

3.3.1.3 read()

The client uses this blocking function to wait and obtain response from the remote server. Not all the request are completed immediat from the server, for the meaning of stream type of protocol. Infact in this protocol, there is a flow for which the complete sequence is defined only at the closure of it3.2.
read() is consuming bytes fom the stream asking to level 4 a portion of them, because it cannot access directly to bytes in Kernel buffer. Lower layer controls the stream of information that comes from the same layer of remove system.

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

RETURN VALUE *Number of bytes read* on success
0 if EOF is reached (end of the stream)
-1 if some error occurs and errno is set appropriately

fd = *Socket File Descriptor* returned by socket().

buf = *Buffer of characters in which it reads and stores info*

count = *Max number of bytes to read* from the file (stream).

So if **read()** doesn't return, this means that the stream isn't ended but the system buffer is empty.
If **read=0**, the function met EOF and the local system buffer is now empty. This helps client to understand that server ended before the connection.

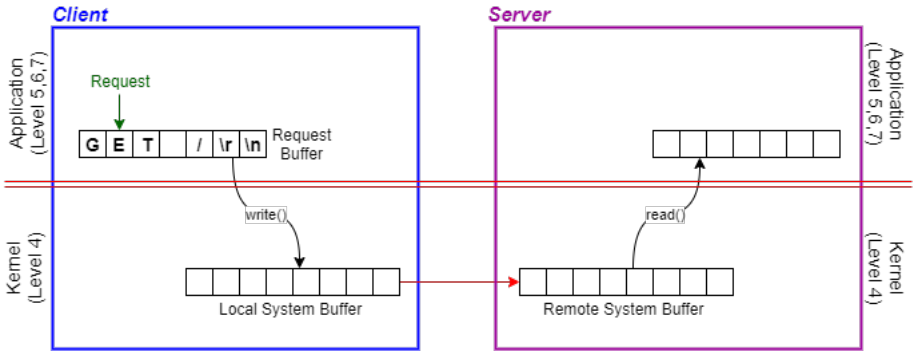


Figure 3.4: Request by the client.

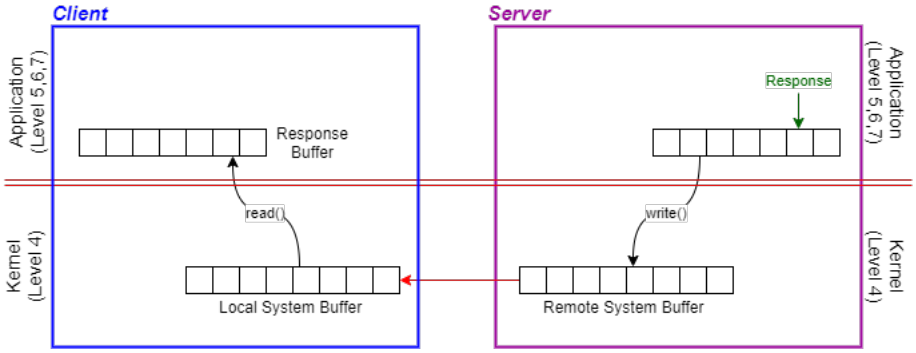


Figure 3.5: Response from the server.

3.3.2 Server

A server is a daemon, an application that works in background forever. The end of this process can be made only through the use of the Operating System.
The server usually uses parallel programming, to guarantee the management of more than one request simultaneously. Hence each process is composed by an infinite loop, as mentioned before.

3.3.2.1 bind()

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

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

RETURN VALUE *0* on success
-1 if some error occurs and errno is set appropriately
(You can check value of errno including <errno.h>).

sockfd = *Socket File Descriptor* returned by socket().

addr = *Reference to struct sockaddr*
sockaddr is a general structure that defines the concept of address.

addrlen = *Length of specific data structure used for sockaddr.*

3.3.2.2 listen()

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

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

The listening socket, identified by **sockfd**, is unique for each association of a port number and a IP address of the server (Figure 3.7).

RETURN VALUE *0* on success
-1 if some error occurs and errno is set appropriately
(You can check value of errno including <errno.h>).

sockfd = *Socket File Descriptor* returned by socket().

backlog = *Maximum length of queue of pending connections*
The number of pending connections for sockfd can grow up to this value.
The normal distribution of new requests by clients is usually Poisson, organized as in Figure 3.6.

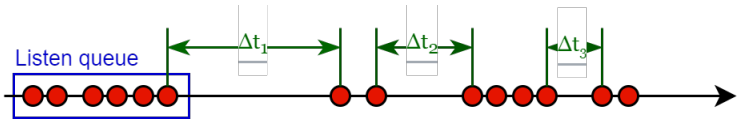


Figure 3.6: Poisson distribution of connections by clients.

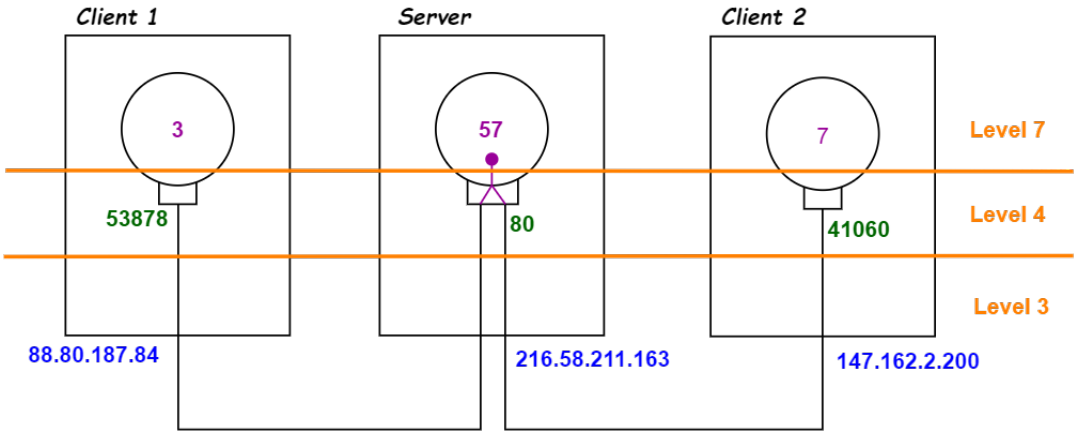


Figure 3.7: listen() function.

3.3.2.3 accept()

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

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

To manage many clients requests, we use the **accept()** function to extablish the connection one-to-one with each client, creating a uniquely socket with each client.
This function extracts the first connection request on the queue of pending connections for the listening socket **sockfd** creates a new connected socket, and returns a new file descriptor referring to that socket. The accept() is blocking for the server when the queue of pending requests is empty (Figure 3.9).
At lower layers of ISO/OSI, the port number and the IP Address are the same identifiers, to which listening socket is associated (Figure 3.8).
The server needs to do a fork after doing the accept(), inside the infinite loop. Hence a new process is created

RETURN VALUE *Accepted Socket Descriptor*
it will be used by server, to manage requests and responses from that specific client.
-1 if some error occurs and errno is set appropriately (You can check value of errno including <errno.h>).

sockfd = *Listen Socket File Descriptor*

addr = *Reference to struct sockaddr*
It's going to be filled by the accept() function.

addrlen = *Length of the struct of addr.*
It's going to be filled by accept() function.
(accept() is used in different cases so it can return different type of specific implementation of struct addr.)

to manage a new request and there is a pair client-worker for each client. So the server can be seen as it would be composed by many servers (Figure 3.10).

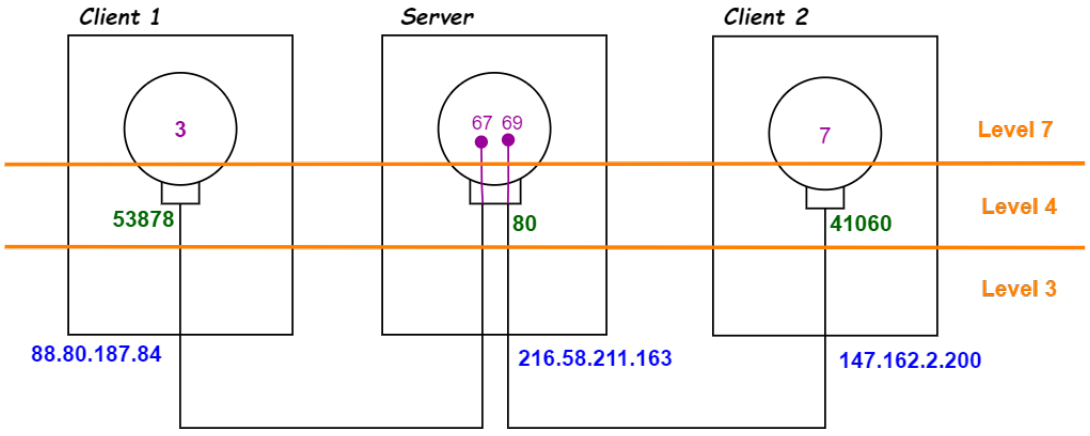


Figure 3.8: accept() function.

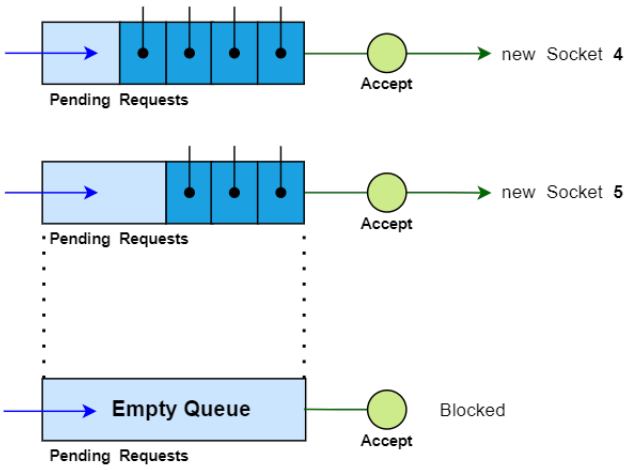


Figure 3.9: Management of pending requests with accept().

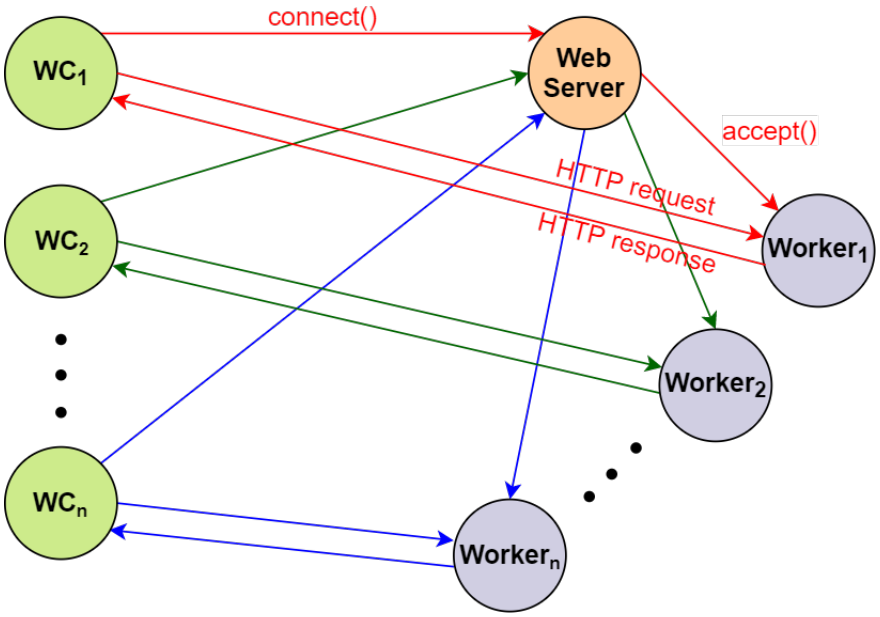


Figure 3.10: connect() and accept() functions in parallel server implementation.

3.4 UDP connection

UDP connection is defined by type **SOCK_DGRAM** as specified in Section 3.2. It's used for application in which we use small packets and we want immediate feedback directly from application. It isn't reliable because it doesn't need confirmation in transport layer.
It's used in Twitter application and in video streaming. **SOCK_DGRAM** is used to read and write directly packets from/to Layer-2, with its header. Layer-2 header is added and removed by the Operating System.
As communication domain, as TCP connection, we can use either **AF_INET** for IPv4 or **AF_INET6** for IPv6. The struct sockaddr, used in this type of connection, is **struct sockaddr_in** like in TCP because of AF_INET domain.

3.5 recvfrom

This function is used to read the whole packet or frame, and only if the size of the buffer, specified as parameter, is lower than the real size of the packet, the function will split the packet and read at first the maximum size available.
Through this function we are going to read the message packet, with format related to the packet format, depending on which layer we are making the call.

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

ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

RETURN VALUE *Number of bytes received* on success
-1 if some error occurs and errno is set appropriately
(You can check value of errno including <errno.h>).

sockfd = *Socket File Descriptor*

buf = *Buffer in which the function will put the message*

len = *Length of the buffer buf*
important to fullfill the buffer in input (usually buf has size equal to the MTU of the network).

flags = *Flags*
added to change the behaviour of the protocol used.

src_addr = *Reference to struct sockaddr*
It's going to be filled by the **recvfrom()** function.

addrlen = *Length of the struct of addr.*
It's going to be filled by accept() function.

3.6 sendto

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

ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

RETURN VALUE *Number of characters sent* on success
-1 if some error occurs and errno is set appropriately
(You can check value of errno including <errno.h>).

sockfd = *Socket File Descriptor*

buf = *Buffer in which the function will get the message*

len = *Length of the buffer buf*
important to read the buffer in input (usually buf has size equal to the MTU of the network)

flags = *Flags*
added to change the behaviour of the protocol used.

dest_addr = *Reference to struct sockaddr*
It's going to be filled by the **recvfrom()** function.

addrlen = *Length of the struct of addr.*

3.7 Lower level connection

Creating a socket, we can also access to lower packet in ISO/OSI model, by selecting other types of communication semantics (Figure 3.2). **SOCK_RAW** is used to read and write directly packets from/to device driver (Layer 1), before adding Layer-2 header. The header needs to be add by us, in writing phase.
Using this communication semantics, we need to use the communication domain **AF_PACKET**. The related socket is duplicated and the user program can access packets, even if it's not working at kernel level. This domain is also used to detect messages in sniffer applications (e.g. Wireshark).
The socket will be created through the following function call (*packet(**7**)*):

```
int packet_socket = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```

The **ETH_P_ALL** guarantees to receive all protocols packets. To obtain the permission from Linux systems, we need to do the following shell command before executing the program. Otherwise the socket won't be created because the operation is not permitted.

```
setcap cap_net_raw,cap_net_admin=eip ./my_exeutable
```

3.7.1 Structure of Layer 2

```
struct sockaddr_ll {
unsigned short sll_family;   /* Always AF_PACKET */
unsigned short sll_protocol; /* Physical-layer protocol */
int            sll_ifindex;  /* Interface number */
unsigned short sll_hatype;   /* ARP hardware type */
unsigned char  sll_pkttype;  /* Packet type */
unsigned char  sll_halen;    /* Length of address */
unsigned char  sll_addr[8];  /* Physical-layer address */
};
```

If we want to talk directly to device driver, we need to specify only two fields:

- **sll_family** = *AF_PACKET*
  the only field common to every struct sockaddr.

- **sll_ifindex** = *index of ethernet interface*
  to obtain it, we can call the following function:

```
#include <net/if.h>
unsigned int if_nametoindex(const char *ifname);
```

RETURN VALUE *Index number of the network interface*
-1 if some error occurs and errno is set appropriately
(You can check value of errno including <errno.h>).

ifname = *Network interface name*

Given in input the name of the network interface (e.g. *"eth0"*), the function returns its related number.

3.8 Usefull functions

3.8.1 Time

3.8.1.1 gettimeofday()

It resturns the number of seconds and microseconds since the Epoch.

```
#include <sys/time.h>
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

RETURN VALUE *0* on success
-1 if some error occurs and errno is set appropriately.
(You can check value of errno including <errno.h>).

tv = *Number of seconds and microseconds since the Epoch*
It will be filled (it can't be NULL)

tz = *Obsolete (normally specified to NULL)*

```
#include <sys/time.h>
struct timeval {
  time_t tv_sec; /* seconds */
  suseconds_t tv_usec; /* microseconds */
};

struct timezone {
  int tz_minuteswest; /* minutes west of Greenwich */
  int tz_dsttime;     /* type of DST correction */
};
```

3.8.1.2 settimeofday()

It sets the number of seconds and microseconds since the Epoch.

```
#include <sys/time.h>
int settimeofday(const struct timeval *tv, const struct timezone *tz);
```

RETURN VALUE *0* on success
-1 if some error occurs and errno is set appropriately.
(You can check value of errno including <errno.h>).

tv = *Number of seconds and microseconds since the Epoch*
It will be filled (it can't be NULL)

tz = *Obsolete (normally specified to NULL)*

3.8.1.3 time()

```
#include <time.h>
time_t time(time_t *tloc);
```

RETURN VALUE *Number of seconds since the Epoch* on success
((time_t) -1) if some error occurs and errno is set appropriately.
(You can check value of errno including <errno.h>).

tloc = *Stored return value (if not NULL)*

3.8.1.4 gmtime()

It converts the calendar time to broken-down time representation, expressed in Coordinated Universal Time (UTC). It may return NULL when the year does not fit into an integer. The return value points to a statically allocated struct which might be overwritten by subsequent calls to any of the date and time functions.

```
#include <time.h>
struct tm *gmtime(const time_t *timep);
```

RETURN VALUE *Pointer to a struct tm* on success
NULL if some error occurs and errno is set appropriately.
(You can check value of errno including <errno.h>).

timep = *Calendar time*

```
#include <time.h>
struct tm {
   int tm_sec;    /* Seconds (0-60) */
   int tm_min;    /* Minutes (0-59) */
   int tm_hour;   /* Hours (0-23) */
   int tm_mday;   /* Day of the month (1-31) */
   int tm_mon;    /* Month (0-11) */
   int tm_year;   /* Year - 1900 */
   int tm_wday;   /* Day of the week (0-6, Sunday = 0) */
   int tm_yday;   /* Day in the year (0-365, 1 Jan = 0) */
   int tm_isdst; /* Daylight saving time */
};
```

3.8.1.5 localtime()

It converts the calendar time to broken-down time representation, expressed relative to the user's specified timezone.

```
#include <time.h>
struct tm *localtime(const time_t *timep);
```

RETURN VALUE *Pointer to a struct tm* on success
NULL if some error occurs and errno is set appropriately.
(You can check value of errno including <errno.h>).

timep = *Calendar time*

3.8.1.6 timegm()

It's the inverse of *localtime()* function. It returns the calendar time (seconds since the Epoch, 1970-01-01 00:00:00 +0000, UTC), taking the input value to be Coordinated Universal Time (UTC).

```
#include <time.h>
time_t timegm(struct tm *tm);
```

RETURN VALUE *Calendar time (seconds since the Epoch)* on success
((time_t) -1) if some error occurs and errno is set appropriately.
(You can check value of errno including <errno.h>).

tm = *Input time structure*

3.8.1.7 timelocal()

It's the inverse of *localtime()* function. It returns the calendar time (seconds since the Epoch, 1970-01-01 00:00:00 +0000, UTC), taking the local timezone into account when doing the conversion.

```
#include <time.h>
time_t timelocal(struct tm *tm);
```

RETURN VALUE *Calendar time (seconds since the Epoch)* on success
((time_t) -1) if some error occurs and errno is set appropriately.
(You can check value of errno including <errno.h>).

tm = *Input time structure*

3.8.1.8 mktime()

It converts a broken-down time structure, expressed as local time, to calendar time representation. The function ignores the values supplied by the caller in the tm_wday and tm_yday fields.

```
#include <time.h>
time_t mktime(struct tm *tm);
```

RETURN VALUE *Calendar time (seconds since the Epoch)* on success
((time_t) -1) if some error occurs and errno is set appropriately.
(You can check value of errno including <errno.h>).

tm = *Input time structure*

3.8.1.9 strptime()

It's the converse of *strftime()*; it converts the character string pointed to by s to values which are stored in the "broken-down time" structure pointed to by tm, using the format specified by format.

```
#define __USE_XOPEN
#include <time.h>
char *strptime(const char *s, const char *format, struct tm *tm);
```

RETURN VALUE *Pointer to the first charater not processed*
NULL if it fails to match all of the format string.

s = *Input string, processed from left to right*
If the input cannot be matched to the format string, the function stops.

format = *character string of field descriptors and text characters*
field descriptor = % character
other character in format must have matching character in s
except whitespace, which matches zero or more whitespaces

```
struct tm {
  int tm_sec;   /* Seconds (0-60) */
  int tm_min;   /* Minutes (0-59) */
  int tm_hour;  /* Hours (0-23) */
  int tm_mday;  /* Day of the month (1-31) */
  int tm_mon;   /* Month (0-11) */
  int tm_year;  /* Year - 1900 */
  int tm_wday;  /* Day of the week (0-6, Sunday = 0) */
  int tm_yday;  /* Day in the year (0-365, 1 Jan = 0) */
  int tm_isdst; /* Daylight saving time */
};
```

3.8.1.10 strftime()

It formats the broken-down time tm according to the format specification format and places the result in the character array s of size max.

```
#include <time.h>
size_t strftime(char *s, size_t max, const char *format, const struct tm *tm);
```

RETURN VALUE *Pointer to the first charater not processed*
NULL if it fails to match all of the format string.

s = *Output string*
If the input cannot be matched to the format string, the function stops.

format = *character string of field descriptors and text characters*
field descriptor = % character
other character in format must have matching character in s
except whitespace, which matches zero or more whitespaces

3.9 Info about files

3.9.1 stat()

It returns information about a file, in the buffer pointed to by statbuf. stat() retrieves information about the file pointed to by pathname. lstat() is identical to stat(), except that if pathname is a symbolic link, then it returns information about the link itself, not the file that it refers to. fstat() is identical to stat(), except that the file about which information is to be retrieved is specified by the file descriptor fd.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
```

RETURN VALUE *Pointer to the first charater not processed*
NULL if it fails to match all of the format string.

s = *Output string*
If the input cannot be matched to the format string, the function stops.

format = *character string of field descriptors and text characters*
field descriptor = % character
other character in format must have matching character in s
except whitespace, which matches zero or more whitespaces

```
struct stat {
  dev_t     st_dev;         /* ID of device containing file */
  ino_t     st_ino;         /* Inode number */
  mode_t    st_mode;        /* File type and mode */
  nlink_t   st_nlink;       /* Number of hard links */
  uid_t     st_uid;         /* User ID of owner */
  gid_t     st_gid;         /* Group ID of owner */
  dev_t     st_rdev;        /* Device ID (if special file) */
  off_t     st_size;        /* Total size, in bytes */
  blksize_t st_blksize;     /* Block size for filesystem I/O */
  blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

  /* Since Linux 2.6, the kernel supports nanosecond
    precision for the following timestamp fields.
     For the details before Linux 2.6, see NOTES. */

  struct timespec st_atim;  /* Time of last access */
  struct timespec st_mtim;  /* Time of last modification */
  struct timespec st_ctim;  /* Time of last status change */

  #define st_atime st_atim.tv_sec  /* Backward compatibility */
  #define st_mtime st_mtim.tv_sec
  #define st_ctime st_ctim.tv_sec
};
```

Chapter 4

Gateway

A gateway is a device that forwards messages from another device, the client, to a second device, the server or another gateway. In the following figures, there are two examples of gateways: Layer-3 gateways (routers in Section 4.2) and Layer-7 gateways (proxy).

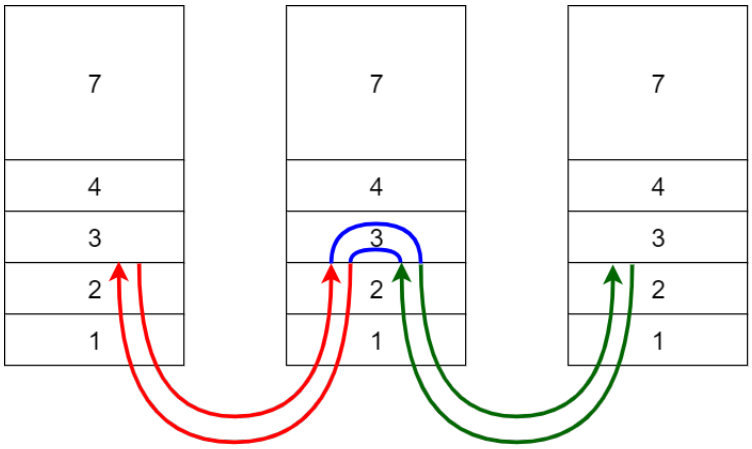


Figure 4.1: Router (Layer-3 gateway).

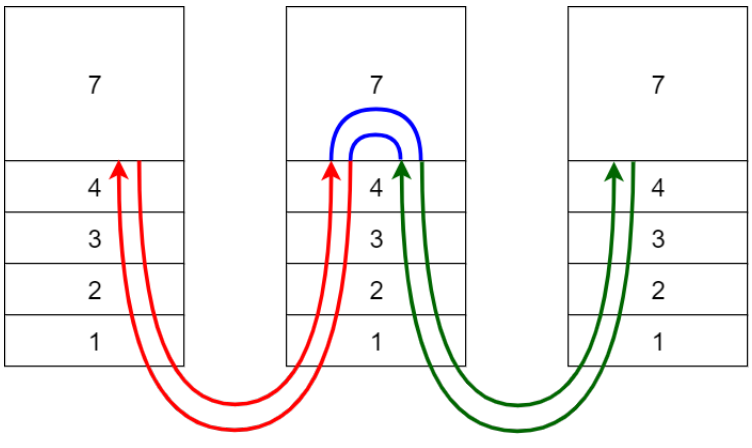


Figure 4.2: Proxy (Layer-7 gateway).

4.1 Proxy

A Layer-7 gateway is also called proxy. It works as an intermediary between two identical protocols (Figure 4.3). Instead of Layer-3 gateways, proxy can also see the full stream of data, analyze HTTP headers and implement new functions. The main possible functions are:

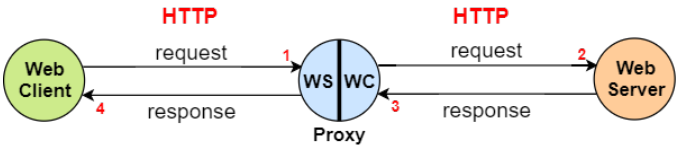


Figure 4.3: Example of proxy use.

- **Caching**
  It's used to reduce traffic directed to the server. The proxy does the most expensive job, managing all the requests of the same page of the server.
  After the request of the page for the first time, the proxy asks the page to the server and then stores in its system, before replying. Hence the next clients requests of the same page will be manage only by proxy because the page was already stored in its system.
  In this case the server needs to manage only a request by proxy and provide a response to proxy.

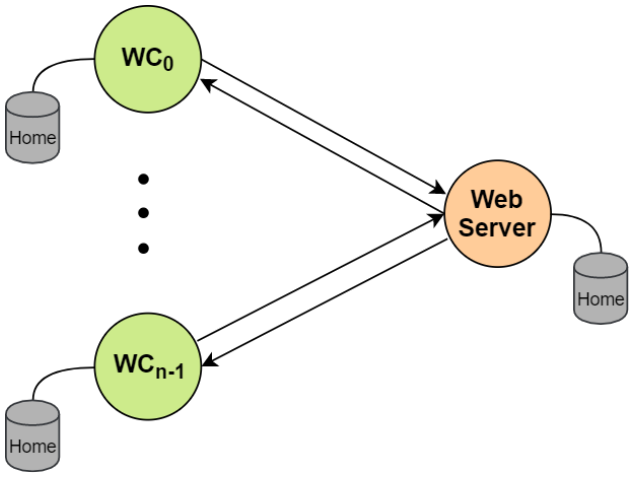


Figure 4.4: Example of caching without proxy.

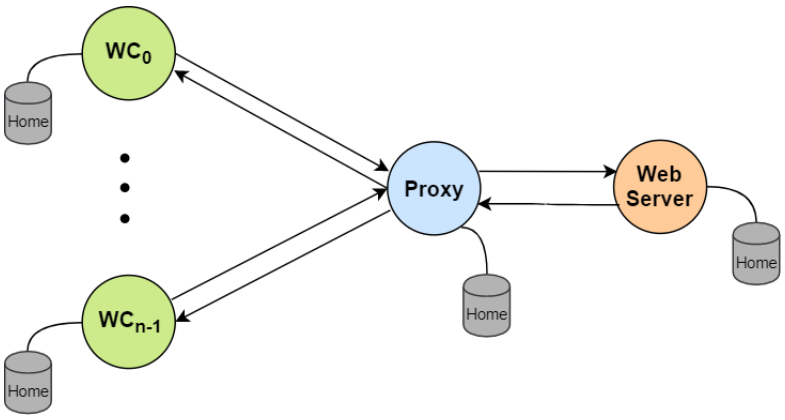


Figure 4.5: Example of caching using proxy.

- **Filtering**
  The proxy can do two actions:
  - **Filtering the requested resource by the client**
    there are many companies that doesn't give access to some services (E.g. no access to Facebook, Youtube, ...).
    We cannot use a filtering approach at lower levels because in some cases clients can access to services through intermediate addresses, different from the one we want to reach. Hence we need to analyze the HTTP request at upper layer.
  - **Filtering the content of the response**
    for parent control approach.

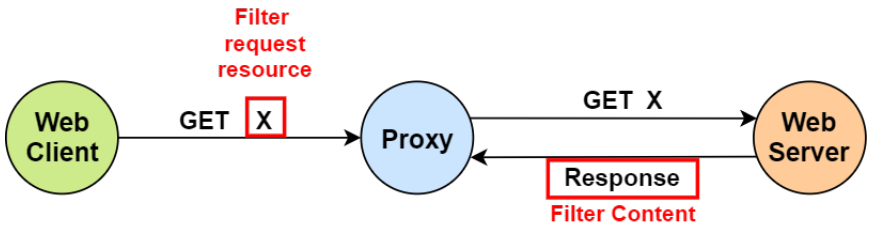


Figure 4.6: Example of proxy filtering.

- **Web Application Firewall (WAF)**
  The proxy is specialized and used to block suspicious requests. This is done by analyzing request content, looking for not secure pattern.
  A possible pattern can be "../" in the path of the resource, that could give access to not accessible part of the File System (injection). Another possible pattern could be a suspicious parameter for a web application to manage SQL database (SQL injection).

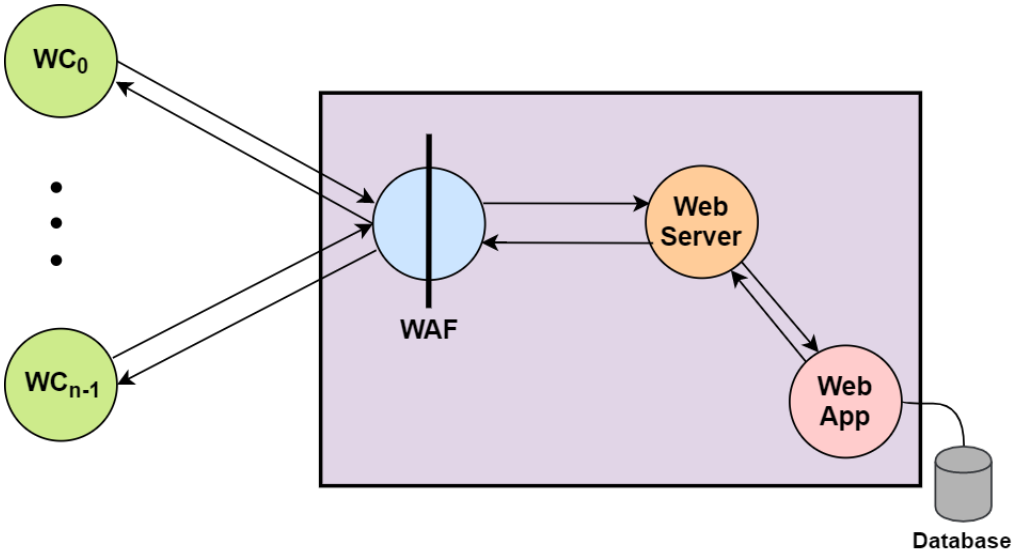


Figure 4.7: Example of WAF use.

- **Load Balancing**
  The proxy is a load balancer for the clients requests to the server.
  There are many servers to manage requests by client. The client makes the request of the web page but in the reality it's talking with the proxy, that manage the request by sending it to a particular server.
  This action is repeated for each client's request. Hence the client thinks that is talking to one server but in reality, the proxy distribute the requests among several servers.

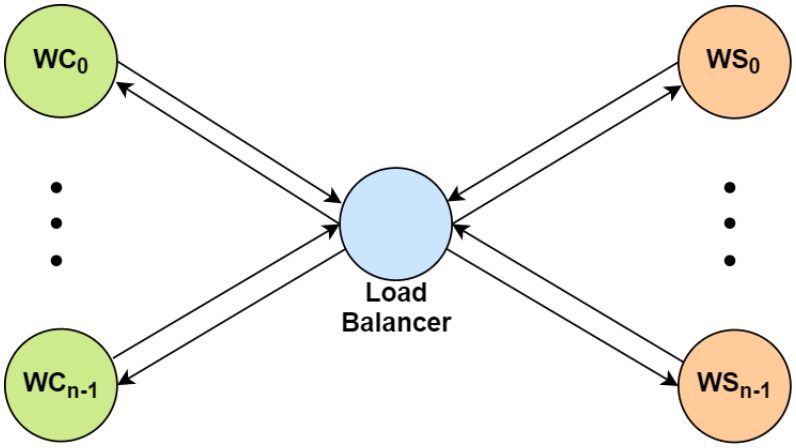


Figure 4.8: Example of load balancing through proxy.

4.2 Router

A router is a device that does two main functions:

1. **Routing**
   it decides on which outbound link send the packet. This decision is based on destination address and its router table (Table 4.1). In each routing table, a network address is associated to an outbound interface, where the packet will be forwarded.
   Each network address is followed by a "/" and a number that defines how many most significant bits of **net mask** are set to 1. The default adddress, that is always in each routing table, is **0.0.0.0**. This one is associated to the interface on which the packet will be sent if no one of the previous messages matches with the one of the destination.
   For each entry of the routing table, the network address is ANDed with its net mask and the IP address, we are looking for, ANDed with that net mask gives us the same result of the first one, the packet is sent to the corresponding interface.
   The default address **0.0.0.0** is associated with a net mask, composed by all 0's. Hence every address, ANDed with this net mask, matches with default address **0.0.0.0**.

Address prefix	Outbound interface
147.162.0.0/*16*	2
88.80.187.0/*24*	4
...	...
0.0.0.0	1

Table 4.1: Example of a routing table.

2. **Switching**
   it sends the packet to the link previously selected.

Each router manages all the incoming packets, storing them in a input **FIFO buffer** (*Standard Service Layer*). By default, if packets arrive too fast to in the buffer, w.r.t. velocity of incoming data processing, new packets are dropped if buffer is already full according to some policy (Figure **??**).
Hence routers has not responsability if some packets are dropped because of it declares it in advance and its goal is to give user the best effort. The behaviour of the router management of the input buffer is based on different policy, according to a goal:

- *To reduce* **latency**
  the packets are sorted by precedence index

- *To reduce* **loss rate**
  dropped packets are the last enetered without R bit set
- *To reduce* **throughput**
  the packets are stored by index, calculated by the router, based on the amount of data transfered from each source/destination in a time unit (e.g. RSUP, virtual clock, MPLS, Stop & GO criteria)

The user cannot set all the possible criteria, because these depend from agreement developed with Service Provider. Hence the Internet Service Provider, if all criteria are set, reset them all before sending packets to Internet.

Chapter 5

Layer 2

It's the layer responsible of sending packets over the network. As it will be explained in Chapter 6 , Layer 3 network disappeared and all local area network are supported by Layer 2. Hence routing isn't needed in the network anymore.
When a smartphone connect to a network, uses a Point to Point Layer 2 connection using LTE/4G/5G, and it's connected to Local Network Area (LAN) using WiFi. Layer 2 supports protocols HDLC, PPP(Point to Point Protocol) in Point to Point connections and Ethernet(IEEE 802.3 802.11) in LAN (Local Area Networks).
Hence Internet Packet passes only through two types of networks: Point to Point link or Local Area Networks.

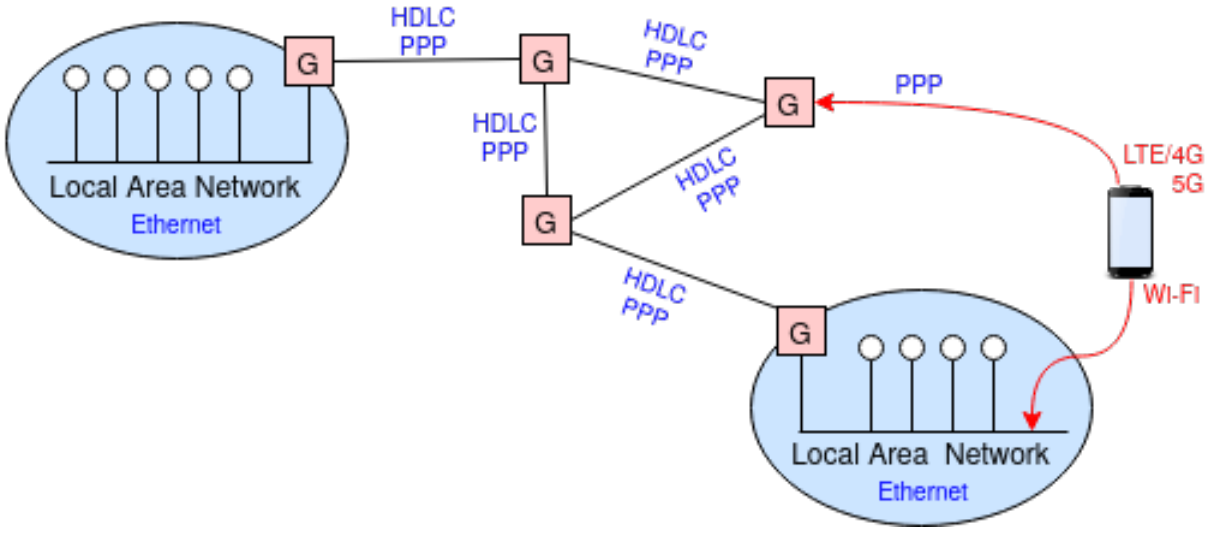


Figure 5.1: Nowadays L2 connections.

5.1 Ethernet

In Ethern protocol, there was a coaxiale cable, long about 1.5 km, on which host interconnect (Figure 5.2). All the hosts electrically shared a bus. In the past hosts ethernet interfaces were connected through a vampire tap junction but now, they are connect to cable using a T-junction (see Figure 5.3 and Figure 5.4). The difference between them is that the first one connects electrically to the cable (connecting it to a cable cut) and the second one is used only in ethernet cables that are physically composed by different cable (segments) and the T-junction is put at intersection of two segments.
The protocol supports Carriege Sense Multiple Access Collision Detection (CSMA/CD), used for coordination between hosts, that it's composed by two strategies:

- **Carrier sense**
  An host can't speak while anyone else is speaking

- **Collision detection**
  the protocol resolves conflicts raised during the contention time. Contention time is the time in which people, that respect first rules, can also go in conflict starting talking together at the same moment.

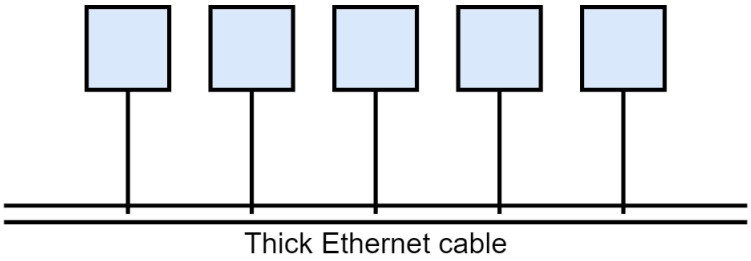


Figure 5.2: Ethernet.

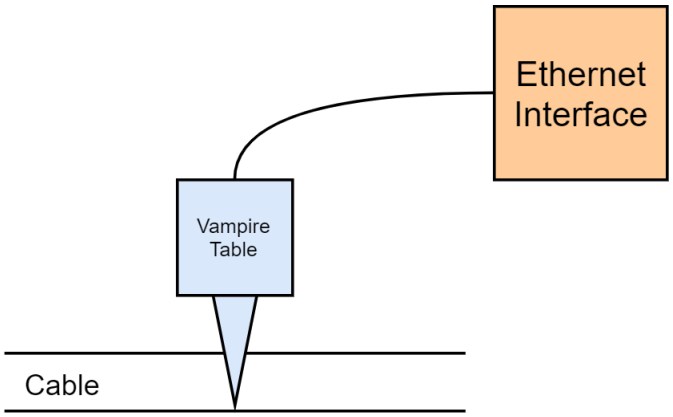


Figure 5.3: Vampire tap.

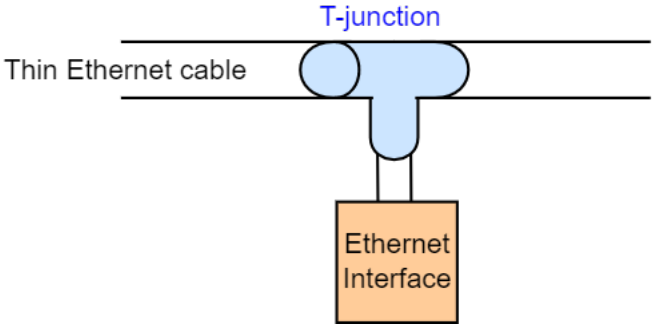


Figure 5.4: T-junction.

In Figure 5.5, N_B detects the collision only when the packet from N_A, arrives to N_B, after the collision with the packet sent by N_A.

The **propagation time (pt)** is the time between the moment in which the host sends the message and the one in which the message arrives to remote host. This time is computed w.r.t. value of light velocity(ideal velocity of packets in Internet) and the absolute distance between the two hosts, that are talking each other.

$$propagation\ time\ (pt) = \frac{absolute\ distance}{light\ velocity}$$

Considering that the absolute distance is about km (10^3 m), the value of the light velocity is 10^8 m/s and the

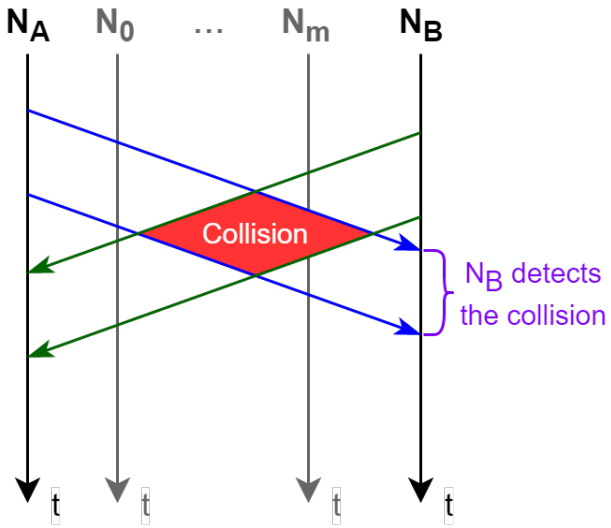


Figure 5.5: Collision detection.

bandwith is about 10^7 bit/sec, we obtain that we can transmitt $10^2 bit$. Hence we could transmit about $10 \div 100$ bytes, but then the number of bytes was standardized to 64 bytes.

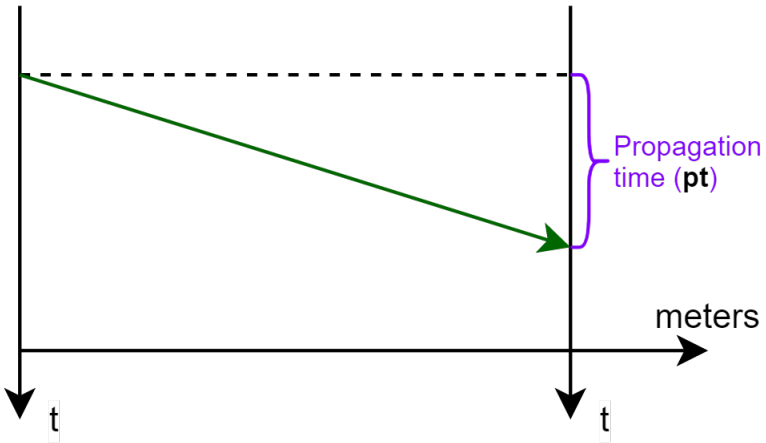


Figure 5.6: Propagation time.

To avoid the collision, when N_B detect the collision, it waits a ranom time to send again the lost previous packet (Figure 5.7). The random time is defined as follows:

$$random\ time = rand() * 2 * pt$$

If there is another collision during this period, the random value *rand()* increases the range in which we can generate a random value. This ranges are defined through this *exponential backoff* sequence:

1) $[0, 1]$
2) $[0, 3]$
3) $[0, 7]$
...
4) $[0, 2^n - 1]$

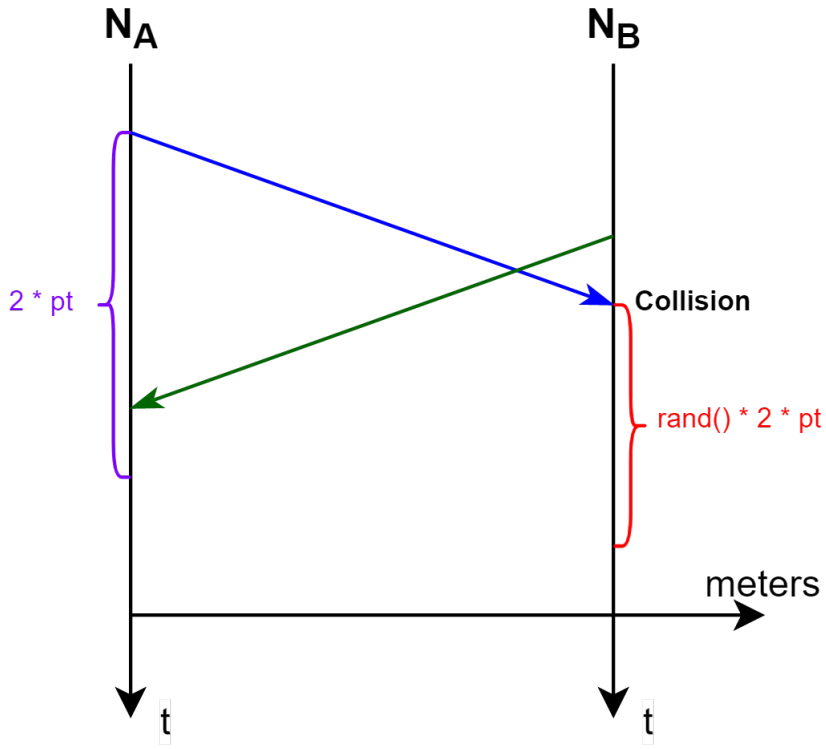


Figure 5.7: Collision avoid.

5.1.1 Ethernet frame

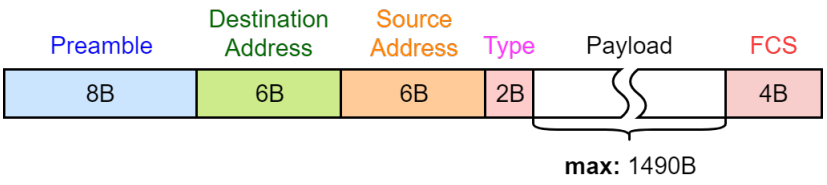


Figure 5.8: Ethernet packet.

- **Preamble**
  synchronization signal *10101010...1010***011** where the last three bits are called **SFD()**.

- **Destination address & Source address**
  MAC (Medium Access Control) addresses, that are Hardware identifiers (broadcast= **ff:ff:ff:ff:ff:ff**).

- **Type**
  type of upper layer protocol used (e.g. Internet Protocol = *0x0800*) [?].

- **Payload**
  payload of the ethernet frame.

- **FCS**
  Frame check sequence (FCS) is a CRC that allows detection of corrupted data within the entire frame as received on the receiver side.

5.1.2 Hub and switches

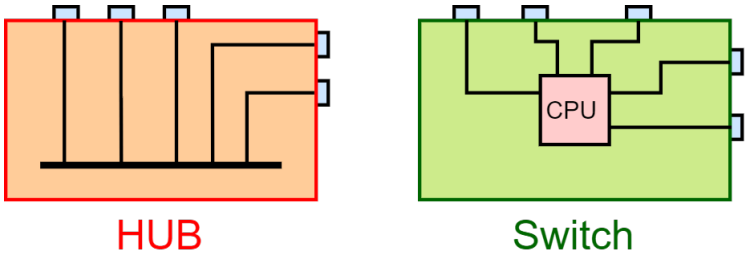


Figure 5.9: Hub and switches.

There two main types of devices, that uses ethernet and creates LANs, are (Figure 5.9):

- **Hub**

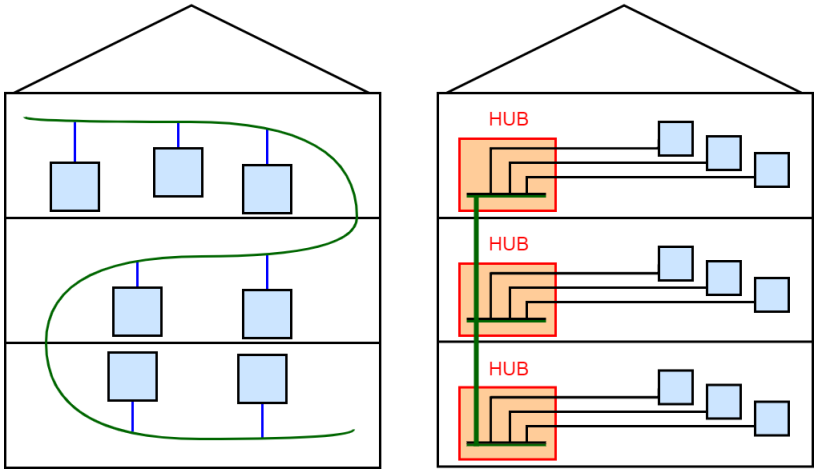


Figure 5.10: Cabled LAN vs LAN with hubs.

  - All the nodes, connected to the hub, receive all packets sent by another node but only destination node considers it. The other ones discard them.
  - Broadcast is very efficient.
  - There is Collision.
  - Network security level is very low.

- **Switch**

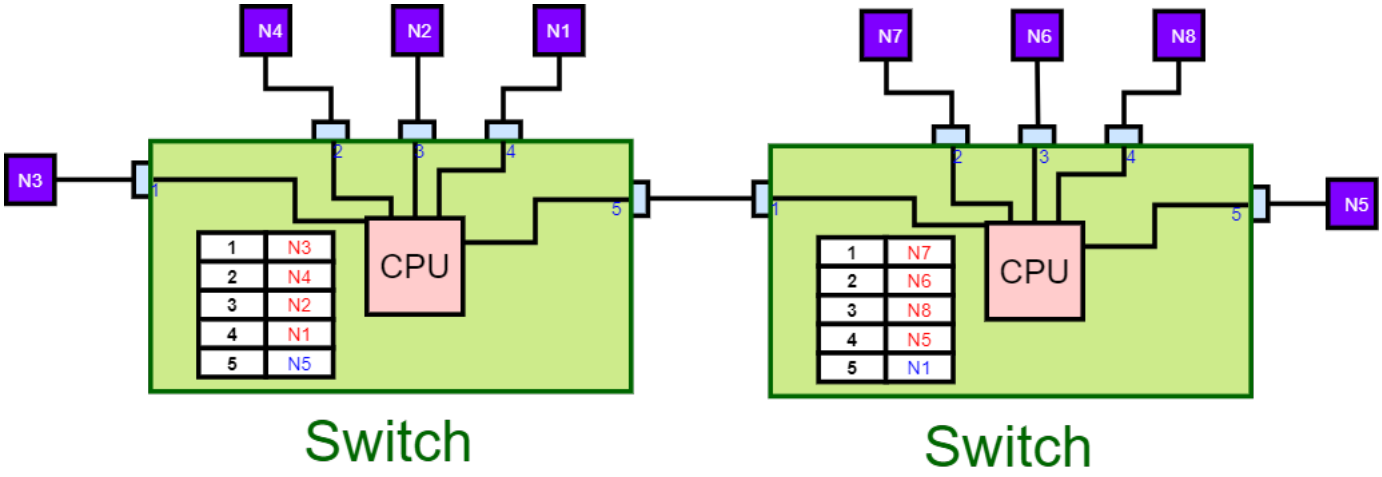


Figure 5.11: Switch connection.

- Only the destination node can see the packets sent by another node to it.
- There aren't collision.
- Broadcast is supported.

In the example of Figure 5.12, there is an aggregate bandwitch of 200 Mbps on a 100 Mbps network.

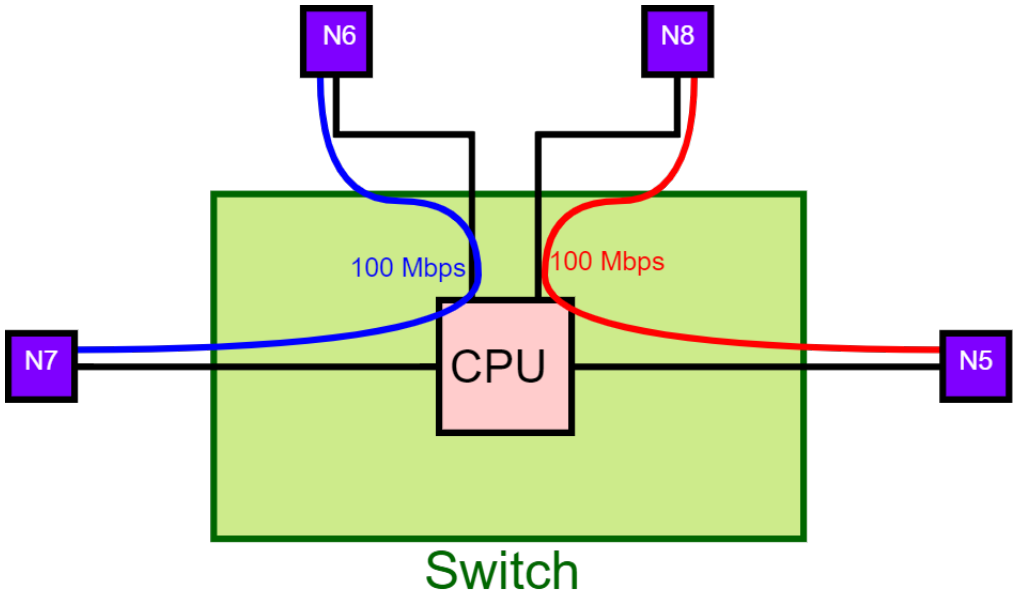


Figure 5.12: Bandwidth switching.

5.1.3 Virtual LAN (VLAN)

Using switches, we can also logical create subnetworks of the hosts connected to the switch. For security reason, the access, to other subnetworks of the hosts connected to the hub, is usually managed through gateways connected to particular ports of the switch. Hence a packet, sent from a virtual network to another one, is sent to that ports to be able reach the final destination.

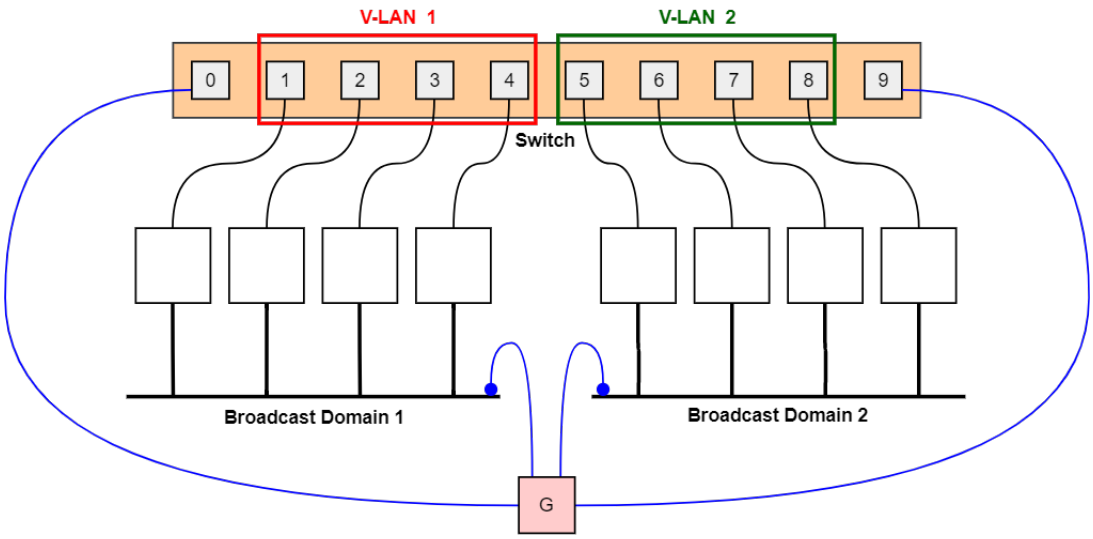


Figure 5.13: VLANs.

It is also possible to create a VLAN over 2 different switches (Figure 5.14). The connection between the two switches is done by adding an Layer-2 or Layer-3 connection. In the second case the connection is called *Lan Emulation Tunneling (VPN)*.

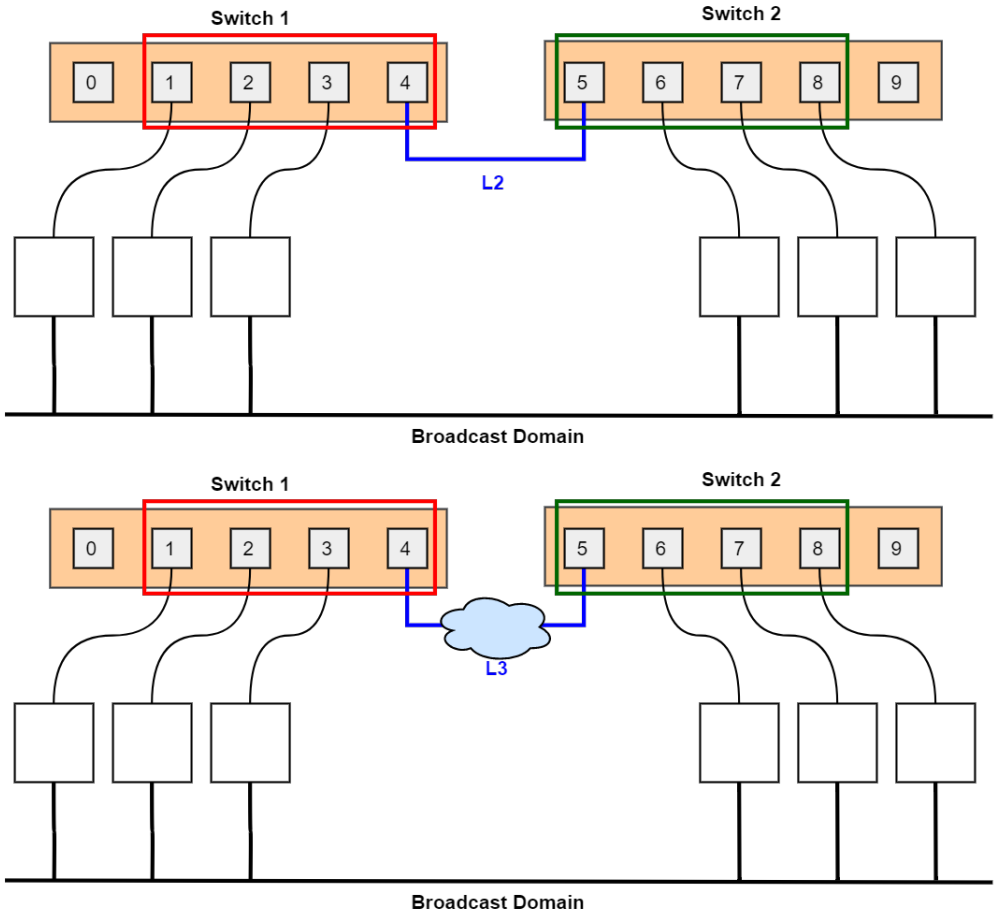


Figure 5.14: VLAN over two switches.

5.1.4 Address Resolution Protocol (ARP)

Using the Ethernet protocol and sending an Internet Protocol packet, the sender needs to know MAC address of remote node. To resolve the IP address of the remote host, we use the DNS protocol. After the IP address is found, we need to resolve the IP address of the destination host into the MAC address of corresponding machine, using the **Address Resolution Protocol (ARP) [?]**.
This method works as follows (Figure 5.16):

```
if( (IP_dest & netmask_src) == (IP_src & netmask_src))
{
    /*
    The source and the destination are in the same network (LAN)
    The answer is sent in broadcast to all the hosts in the network, specifying
    the IP_dest and the host that has the specific IP_dest, replies with its
    MAC_dest

    Then there will be a new packet, sent to [IP_dest, MAC_dest] machine
    (example of this packet in the Figure 6.15)
    */
}
else
{
    /*
    The source and the destination are in different networks (LANs)
    The answer is sent in broadcast, from H_src, asking for the MAC_gat of
    the host in the LAN with with IP_gat

    Then knowing it, it will be sent a new packet to the specific gateway
    host MAC_host but specyfing IP_dest
    (example of this packet in the Figure 6.15)
    */
}
```

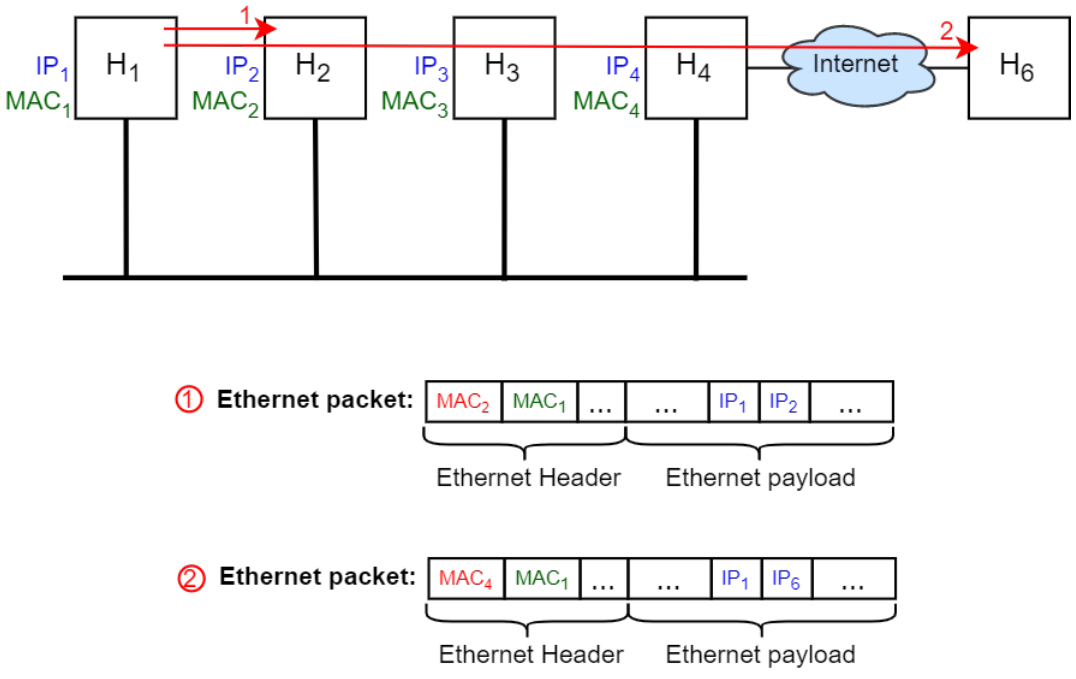


Figure 5.15: ARP.

5.1.4.1 ARP message format

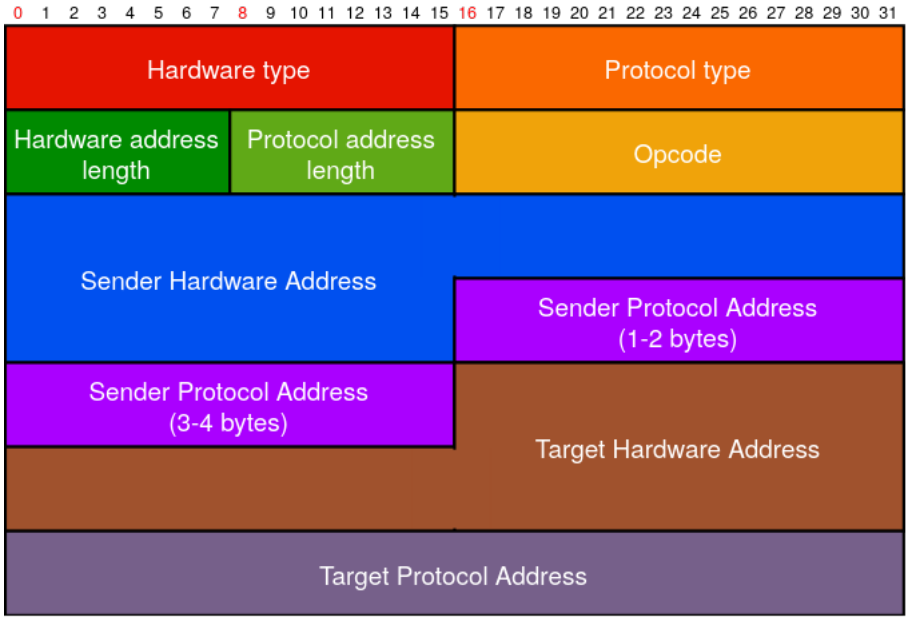


Figure 5.16: ARP message format.

- **Hardware type**
  Hardware type (*0x0001*=Ethernet protocol).

- **Protocol type**
  Protocol type (*0x0800*=Internet protocol).

- **Hardware Address Length**
  Length of Hardware Address in bytes (*6*= MAC address).

- **Protocol Address Length**
  Length of Protocol Address in bytes (*4*= IP address).

- **Opcode**
  code representing the type of ARP message.

0x01	ARP request
0x02	ARP reply
0x03	RARP request
0x04	RARP reply

  RARP protocol works as ARP but it's used to obtain IP address from the MAC address. This is usually used trying to connect to wireless networks. In this case, the user needs to have a specific IP address to connect to Internet and through ARP he can obtain it.
  Today RARP protocol is not used anymore because we use DHCP, an evolution of RARP.

- **Sender Hardware Address**
  Hardware Address of whom sends ARP message.

- **Sender Protocol Address**
  Protocol Address of whom sends ARP message.

- **Target Hardware Address**
  Hardware Address we want to obtaint through ARP or Hardware address we want to solve through RARP (*all zeros* in ARP request).
- **Target Protocol Address**
  Protocol Address that we want to solve or protocol Address we want to obtain through RARP (*all zeros* in ARP request).

Chapter 6

Internet Protocol

The Internet protocol was the result of research job made by american Department of Defence (DoD). *Internet* means Inter-networks communication and was designed for use of interconnected systems of packet-switched computer communication networks. The only things in common between the networks is the packet architecture. Today the Internet Protocol is the only one yet used in Layer 3. The Internet Protocol provides transmission of blocks of data called datagrams, from sources to destinations, where sources and destinations are hosts identified by fixed length addresses [**?**].
The two main functions, that Internet Protocol needs to provide, are:

1. **Definition of unified addresses (Section 6.2)**
2. **Fragmentation (Section 6.3)**

The creation of Internet Protocol comes from the needs of interconnection between networks (Figure 6.1). Each network has its own protocol and it's composed by serveral devices, connected each other. The terminal devices of a network are the hosts and they can talk to others in the net through routers.
The new devices added with the invention of Internet Protocol were the Gateways, devices similar to routers that also translate protocols of different networks. The links inside the network (that connects routers and hosts) work on Layer 3 and the links between gateways work as Layer 2 networks, that doesn't required routing function.
Nowadays, networks are almost local so the gateways work mostly as routers. In fact, the routers don't exist as their definition tells (Figure 6.2). The routing mechanism is no more done at Layer-3 but at Layer-2.
Ping is the most known service of Internet Protocol.

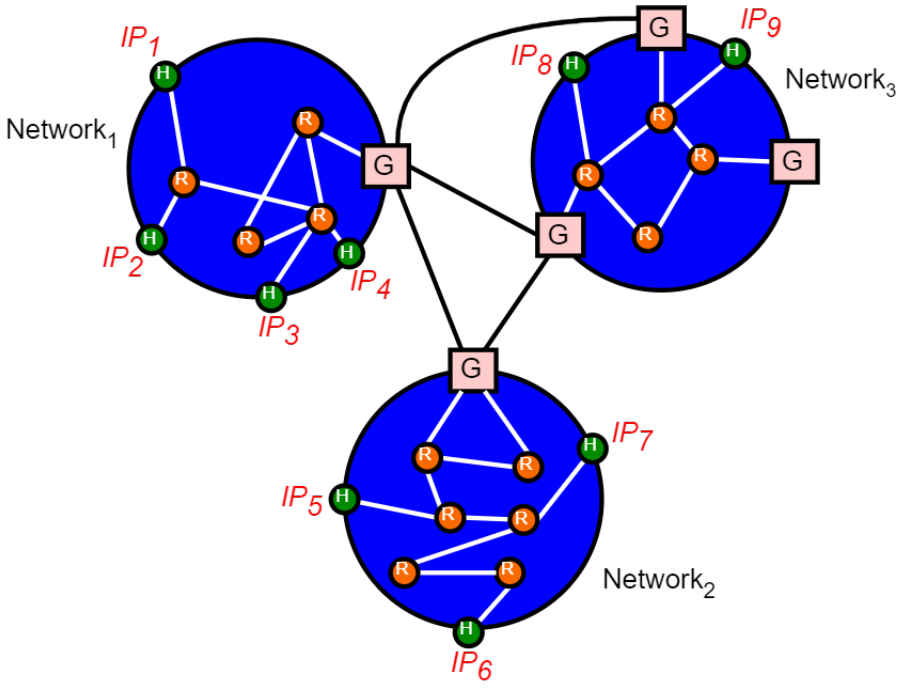


Figure 6.1: Internet structure.

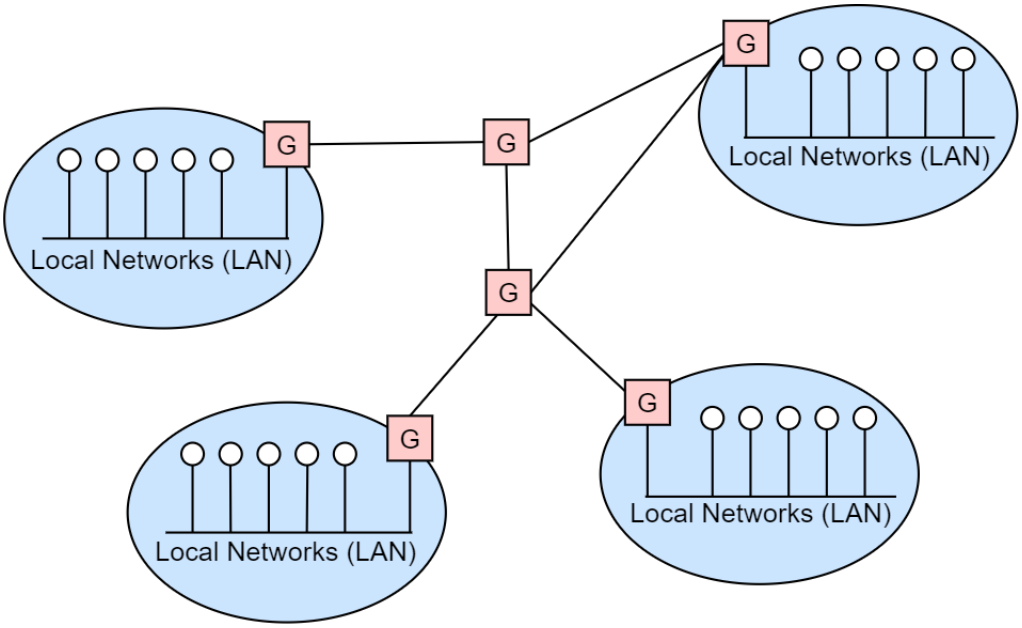


Figure 6.2: LAN structure.

6.1 Terminology

- **Round Trip Time (RTT)**
  time needed from network to send the packet and receive the response packet
- **Delay**
  passed time before the true service
- **Bit rate (Bandwidth)**
  amount of Bit/s or Bytes/s of the network
- **Throughput**
  amount of data/s that I can really transmit
- **Relaibility**
  capacity of being reliable and losing few packets. It's related to inverse of:

$$loss\ \ rate = \frac{\#\ lost\ packets}{\#\ sent\ packets}$$

6.2 IP address

To send packets among different networks, we need to identify gloabally the destination host and IP address was designed to solve this problem. The IP addresses are 32 bits numbers. They are commonly represented as a set of 4 numbers separated by a point and each of them is the decimal representation of the corresponding byte in the IP address.
An IP address can be divided into two parts: Network part and Host part. In the past, the IP addresses were classified by three main classes, based on the size of their Network part: *Class A, Class B, Class C* (Figure 6.3).
This classification of addresses in this way isn't very efficient because this cannot manage well addressing of large number of small networks or small number of large networks.
To do it it was introduced the Net Mask, a bit mask composed by a sequence of 1's followed by 0's, that permits us to define the parts of an address of whatever dimension we want (Figure 6.4). This is useful also to create subnetworks of a given set of hosts (Figure 6.5).
There are also two special addresses:

- **Network address (no hosts)**
  Host part = *0...0000*
- **Broadcast address (all hosts in the network)**
  Host part = *1...1111*

Hence to give an address to each endpoint of a **Point To Point** link, we need to use at least an Host part of 2 bits (Figure 6.6).

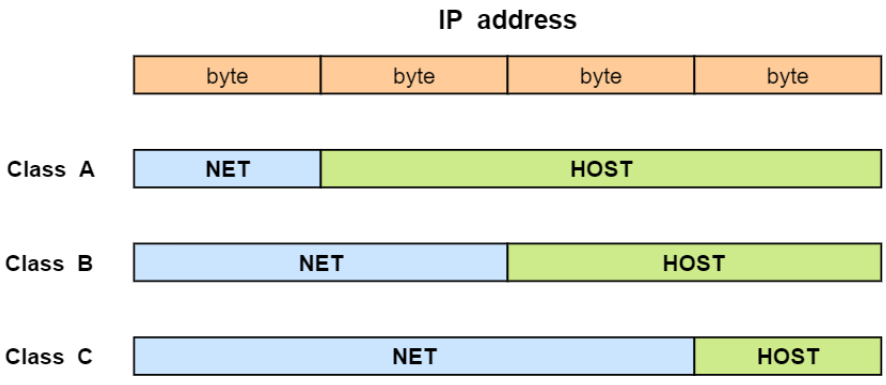


Figure 6.3: IP classes.

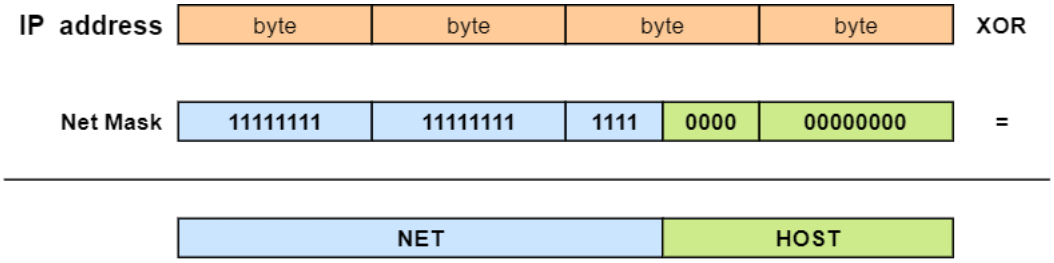


Figure 6.4: Example of netmask use.

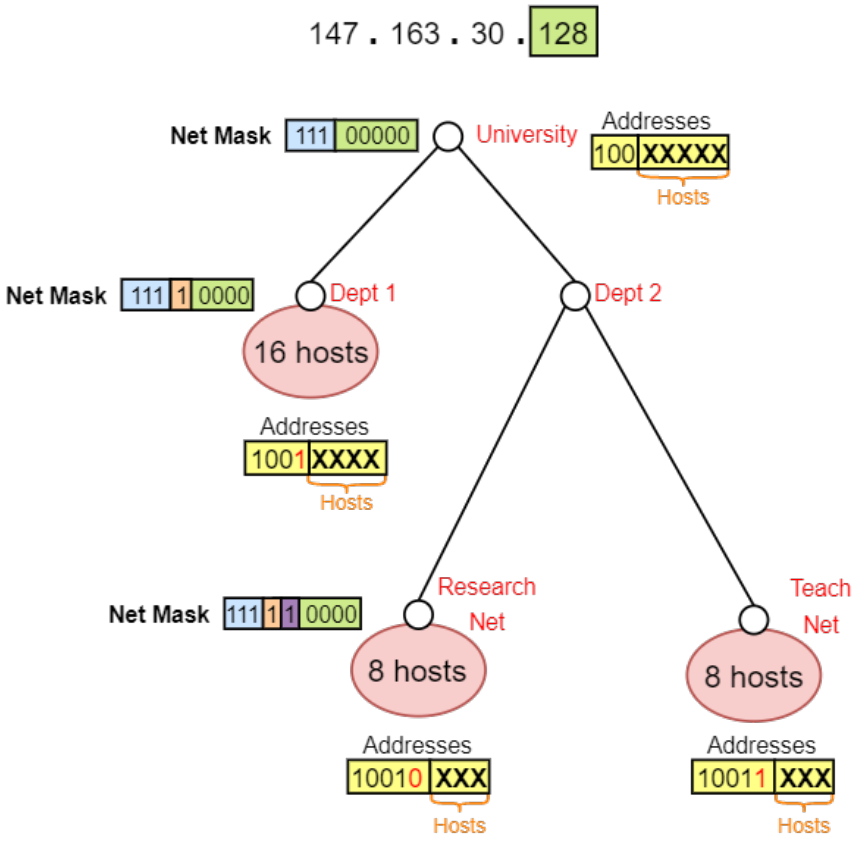


Figure 6.5: Example of subnetworks structure.

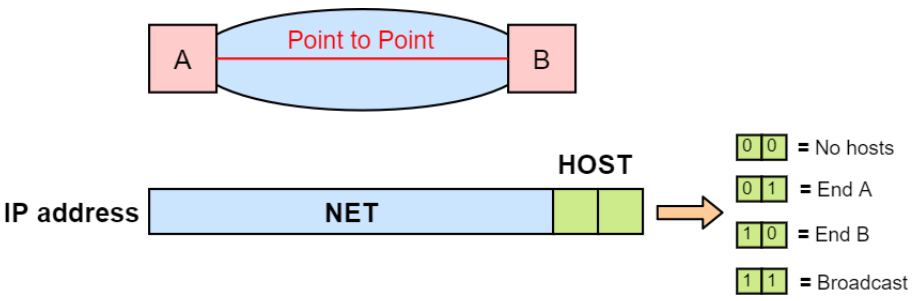


Figure 6.6: Example of Point to Point connection network.

6.3 Fragmentation

In each network, the IP information is embedded in a Layer 3 packet that respects protocol of the network in which it is. Then when the packet reach a gateway, its IP info is removed from the packet and encapsulated in a Layer 2 packet, to be sent to another network (Figure 6.7). Each IP packet is also called **Datagram**.
Each network is defined by a Maximum Transfer Unit (MTU), that defines the maximum size of each Layer 3 packet inside the network. Hence, if the IP information, that reach a gateway of the network, is larger than MTU, the gateway reduces its size (Figure 6.8).
If a packet pass through many networks and their MTUs are very different, using datagrams, we are sure that the packets won't arrive as in the same order in which they are sent. The reason why this happens is that they are sent without the use of a stream. To manage this problem, when the gateway creates a packet, this stores the first index of the sequence of the bytes of the original IP information.
The last packet, that composed initial IP message, has the flag **More Fragments(MF)** set to 0. This information with the knowledge of the length and the first byte index of the last packet, permits to define the length of the original message, whenever it arrives. Each packet can fit easly in the buffer of the gateway receiver (Figure 6.9).

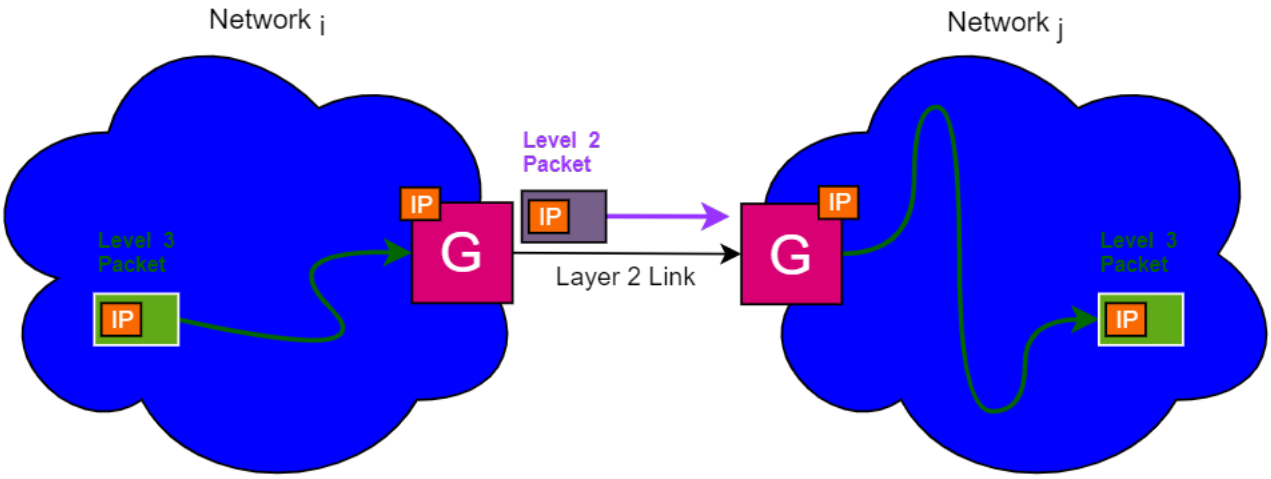


Figure 6.7: Example of encapsulation of IP packet.

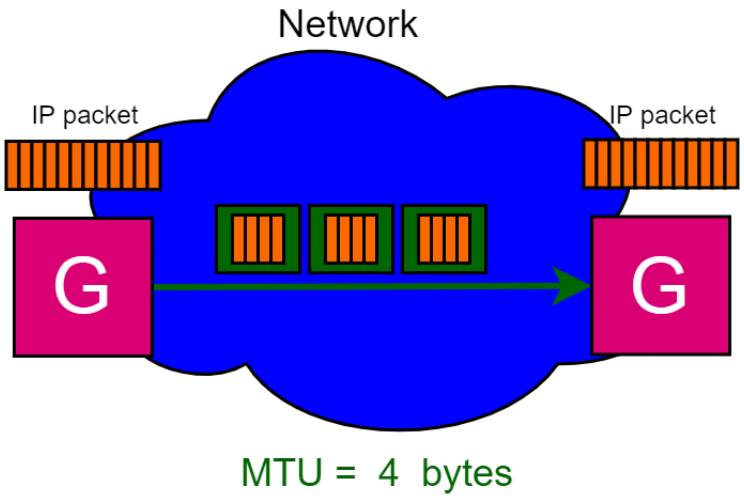


Figure 6.8: Example of fragmentation.

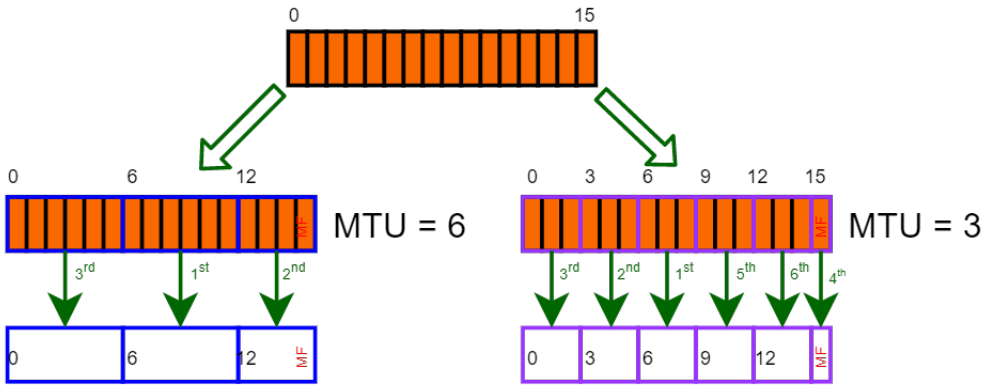


Figure 6.9: Example of fragment labeling.

6.4 Internet Header Format

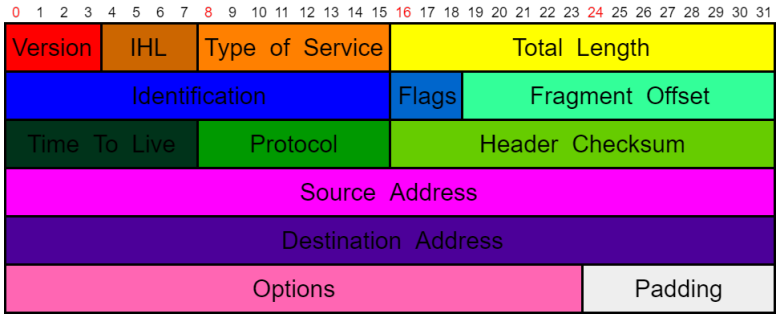


Figure 6.10: Internet header format.

The content of the internet header is (Figure 6.10):

- **Version** format of the internet header
- **IHL**
  length, measured in words of 32 bits, of the internet header (minimum value = 5)
- **Type of Service**
  parameters of the Quality of Service (QoS) desired (Figure 6.12). Bits **6-7** are reserved for future use.

Figure 6.11: Type of service field.

	Delay (D)	Throughput(T)	Relaibility (R)
0	Normal	Normal	Normal
1	Low	High	High

Table 6.1: Bits 3,4,5 of Type of Service.

111	Network Control
110	Internetwork Control
101	CRITIC/ECP
100	Flash Override
011	Flash
010	Immediate
001	Priority
000	Routine

Table 6.2: Precedence of Type of Service.

- **Total Length**
  length, measured in octets, including internet header and data.
  This field allows the length of a datagram to be up to 65,535 octets. Such long datagrams are impractical for most hosts and networks. All hosts must be prepared to accept datagrams of up to 576 octets (whether they arrive whole or in fragments). It is recommended that hosts only send datagrams larger than 576 octets if they have assurance that the destination is prepared to accept the larger datagrams.

- **Identification**
  an identifying value assigned by the sender to aid in assembling the fragments of a datagram.
  It's a random number generated by host while creating the packet, that is different from numbers of all other packets.

- **Flags**
  varius control flags. The bit 0 is reserved and must be 0.

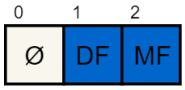


Figure 6.12: Flags.

	Don't Fragment (DF)	More Fragments (MF)
0	May Fragment	Last Fragment
0	Don't Fragment	More Fragments

Table 6.3: DF and MF flags.

If DF set and a packet that arrives to a network should be divided in smaller fragments, it's dropped.

- **Fragment Offset**
  This field indicates where in the datagram this fragment belongs (position of the fragment in the original long packet).
  The fragment offset is measured in units of 8 octets (64 bits). The first fragment has offset zero.
  It's computed starting from initial position in the packet.

- **Time to Live**
  maximum time (number of forward for the packet) the datagram is allowed to remain in the internet

system.
This counter is set by host that generated the packet. Every node in the network (routers, switches), that process the packet, decrements the value of this field.
When a node, decrementing this field, reaches zero value for Time To Live, it drops the packet immediately. Time To Live prevents that a packet stays in the network too much time compromising infrustructure efficiency.

- **Protocol**
  the next level protocol (Layer 4) used in the data portion of the internet datagram. In general it's called ULP (Upper Layer Protocol). This is useful and was done also at upper layer, using port numbers, because it's a way to communicate future use to upper layer. This field is the upper layer protocol type (**/etc/protocols** on UNIX) and it's used by Operating System to understand to which module send a specific part of the packet. You can also find them in IANA site [**?**].

- **Header Checksum**
  a checksum on the header only.

  How to compute it
  The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero. The two main operation used in its computation are:

  – **One's complement sum($\oplus$)**
    two words of 16 bits are summed up, bit by bit, and the last carry is summed up to the previous result. The following example shows how to sum two number with this operator:

$$\begin{array}{rl} 10110 \ldots 10 & + \\ 01101 \ldots 11 & = \\ \hline 00100 \ldots 01 & + \\ \text{carry: } 1 & = \\ \hline 00100 \ldots 10 & \end{array}$$

  – **Ons's complement**
    the value of each bit, inside the result of 16 bit sum of all the words, change their values.

$$\begin{array}{r} 00100 \ldots 10 \\ \hline 11011 \ldots 01 \end{array}$$

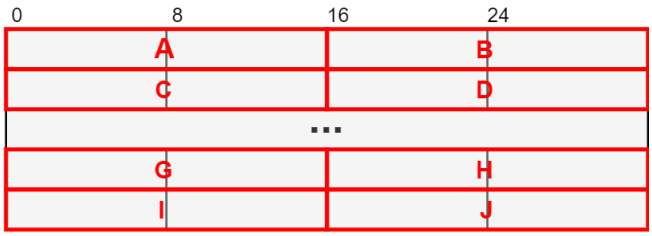


Figure 6.13: Words of payload evaluated in checksum.

$$Checksum = \tilde{}(A \oplus B \oplus C \oplus D \oplus ... \oplus A \oplus B \oplus C \oplus D \oplus)$$

This algorithm is very simple but experimental evidence indicates it works. Nowadays, it's quite always used CRC procedure.

- **Source Address**
  the source IP address

- **Destination Address**
  the destination IP address
- **Options**
  it'svariable and it may appear or not in datagrams. They must be implemented by all IP modules (host and gateways).
  What is optional is their transmission in any particular datagram, not their implementation.

Chapter 7

ICMP

ICMP (Internet Control Message protocol) messages are embedded into IP datagrams [?]. ICMP can also be seen as a protocol that makes use of IP.

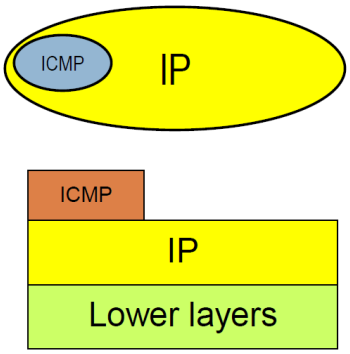


Figure 7.1: How ICMP is embedded in IP datagrams.

The main controls, made by ICMP, are:

- **Error management (passive)**
  - Destination unreachable
  - Time expired (TTL or fragment reassembly timer)
  - Data inconsistency
  - Flow control
- **Active mode**
  Echo + Echo Reply (ping Unix)

In the IP header, the field protocol takes value 1 and indicates that the payload is an ICMP message.

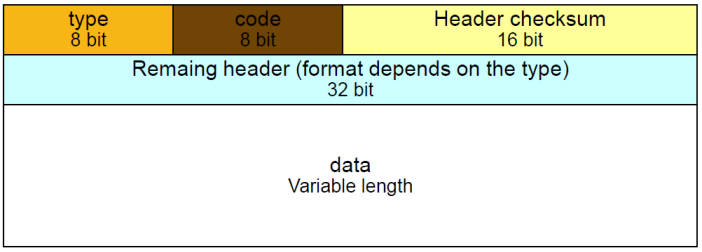


Figure 7.2: Format of ICMP message.

0	Echo reply
3	Destination unreachable
4	Source Quench
5	Redirect (change a route)
8	Echo request
11	Time exceeded
12	Parameter problem
13	Timestamp request
14	Timestamp reply
17	Address mask request
18	Address mask reply

Table 7.1: Type values.

Other header fields depend on the type of message that must to be generated.

7.1 Main rules of ICMP error messages

- No ICMP error message will be generated in response to a datagram carrying an ICMP error message
- No ICMP error message will be generated for a fragmented datagram that is not the first fragment
- No ICMP error message will be generated for a datagram having a multicast address
- No ICMP error message will be generated for a datagram having a special address such as 127.0.0.0 or 0.0.0.0.

NOTE: *No all routers generate ICMP messages.*

7.2 Types of ICMP messages

7.2.1 Echo

Echo-request and Echo-reply are used to check the reachability of hosts and routers.
Upon receiving an Echo-request, the ICMP entity of a device immediately replies with Echo reply.

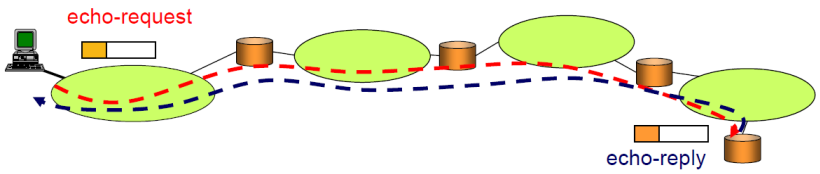


Figure 7.3: ECHO requests and replies in practice.

Type: $\left|\begin{array}{l}\Rightarrow 8 \text{ request}\\ \Rightarrow 0 \text{ reply}\end{array}\right.$

Code: $\Rightarrow 0$

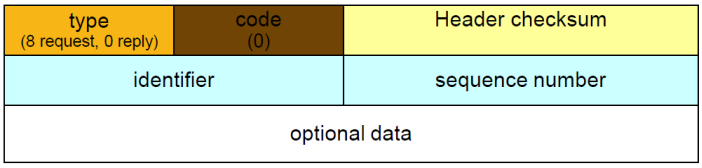


Figure 7.4: Format of ECHO message.

Other important fields of Echo messages are:

- **Identifier**
  Each **Echo** message has an identifier, defined in the **Echo request**, and replicated in the **Echo reply**.

- **Sequence number**
  Consecutive requests may have the same identifier and change from others for sequence number only. The sequence number is used to measure the RTT and count the number of lost bytes.

- **Optional data**
  The sender can add **Optional data** to the request message. The data will be replicated in the reply message.

The payload of Echo (IP datagram) is used to check the capacity of a link (RTT is bigger if the link has small bitrate).

7.2.2 Destination unreachable

When a packet is dropped, an error message is returned, through ICMP, to the source.

Type: $\Rightarrow 3$

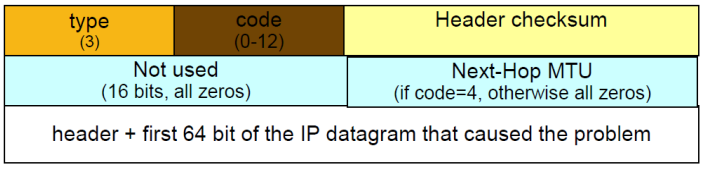


Figure 7.5: Destination unreachable message format.

The "code" field of the ICMP message refers to the type of error that has generated the message.

Code	Description	References
0	Network unreachable error.	RFC 792
1	Host unreachable error.	RFC 792
2	Protocol unreachable error. Sent when the designated transport protocol is not supported.	RFC 792
3	Port unreachable error. Sent when the designated transport protocol is unable to demultiplex the datagram but has no protocol mechanism to inform the sender.	RFC 792
4	The datagram is too big. Packet fragmentation is required but the DF bit in the IP header is set.	RFC 792
5	Source route failed error.	RFC 792
6	Destination network unknown error.	RFC 1122
7	Destination host unknown error.	RFC 1122
8	Source host isolated error. (Obsolete)	RFC 1122
9	The destination network is administratively prohibited.	RFC 1122
10	The destination host is administratively prohibited.	RFC 1122
11	The network is unreachable for Type Of Service.	RFC 1122
12	The host is unreachable for Type Of Service.	RFC 1122
13	Communication Administratively Prohibited. Administrative filtering prevents a packet from being forwarded.	RFC 1812
14	Host precedence violation. The requested precedence is not permitted for the particular combination of host or network and port.	RFC 1812
15	Precedence cutoff in effect. The precedence of datagram is below the level set by the network administrators.	RFC 1812

Table 7.2: Code values.

7.2.3 Time exceeded

It's generated when some packets are missing or don't reach the destination.

Type: $\Rightarrow 3$

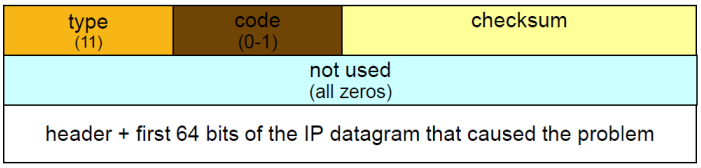


Figure 7.6: Time exceeded message format.

The main problems, that generate this message, are:

Code	Problem
0	Generated by a router when it decreases the TTL to 0
	Returned to the source of the IP datagram
1	Generated by the destination, when some fragments are missing, after the fragment reasembly timer expires

7.2.4 Parameter problem

It's generated when there are some wrong formats or unknown options.

Type: $\Rightarrow 12$

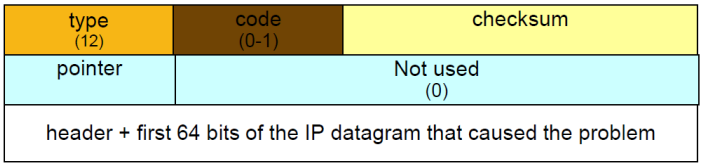


Figure 7.7: Format of Parameter problem message.

The main problems generated by this message are:

Code	Problem
0	If the header of an IP datagram contains a malformed field (violate format)
1	Used when an option is unknown or a certain operation cannot be carried out

7.2.5 Redirect

It's generated by a router to require the source to use a different router

Type: $\Rightarrow 5$
Code: $\Rightarrow 0-3$

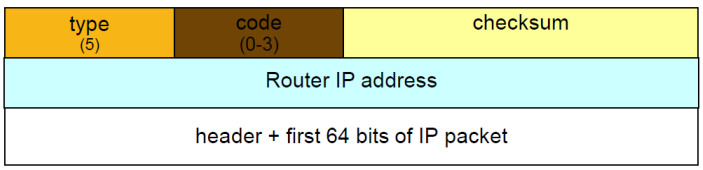


Figure 7.8: Format of Redirect message.

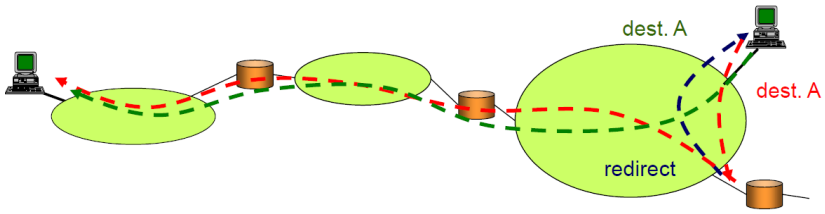


Figure 7.9: How Redirect messages are used.

7.2.6 Timestamp request e reply

It's used to exchange clock information between source and destination.

Type: $\Rightarrow 13$ request
$\Rightarrow 14$ reply

Code: ⇒ 0

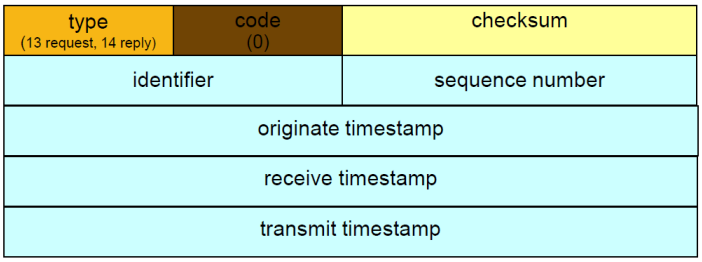


Figure 7.10: Format of Timestamp request and reply.

- **Originate timestamp**
  inserted by the source
- **Receive timestamp**
  inserted by the destination right after receiving the ICMP message
- **Transmit timestamp**
  inserted by the destination just before returning the ICMP message

7.2.7 Address mask request and reply

It's used to ask for the netmask of a router/host.

Type: ⇒ 17 request
⇒ 18 reply

Code: ⇒ 0

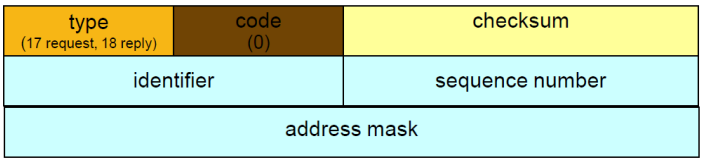


Figure 7.11: Format of Address mask request and reply.

- **Address mask**
  In the request message, it's void and it is populated by the device that replies to the request

Chapter 8

Transport layer

8.1 UDP (User Data protocol)

This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks [?]. This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP).

8.1.1 UDP packet format

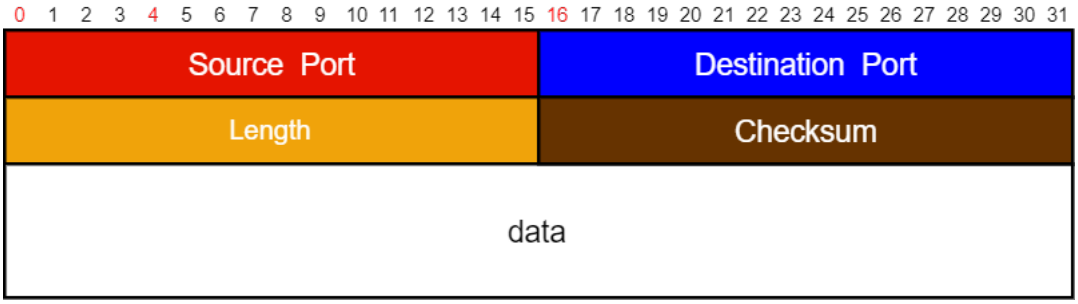


Figure 8.1: UDP packet format.

- **Source Port** (16 bits)
  The source port number
- **Destination Port** (16 bits)
  The destination port number
- **Length (16 bits)** The length in octets of this user datagram including this header and the data
- **Checksum** (16 bits)
  The checksum of information from the IP header, the UDP header, and the data, padded with zero octets at the end (if necessary) to make a multiple of two octets.
  The pseudo header, conceptually prefixed to the UDP header, contains the source address, the destination address, the protocol, and the UDP length. This information gives protection against misrouted datagrams. This checksum procedure is the same as is used in TCP.
  If the computed checksum is zero, it is transmitted as all ones (the equivalent in one's complement arithmetic). An all zero transmitted checksum value means that the transmitter generated no checksum (for debugging or for higher level protocols that don't care).

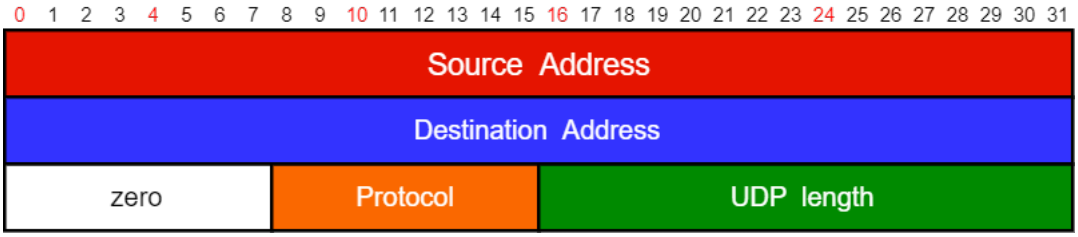


Figure 8.2: Pseudo header.

8.2 TCP (Transmission Control protocol)

The Transmission Control Protocol (TCP) is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such networks [?].

8.2.1 TCP packet format

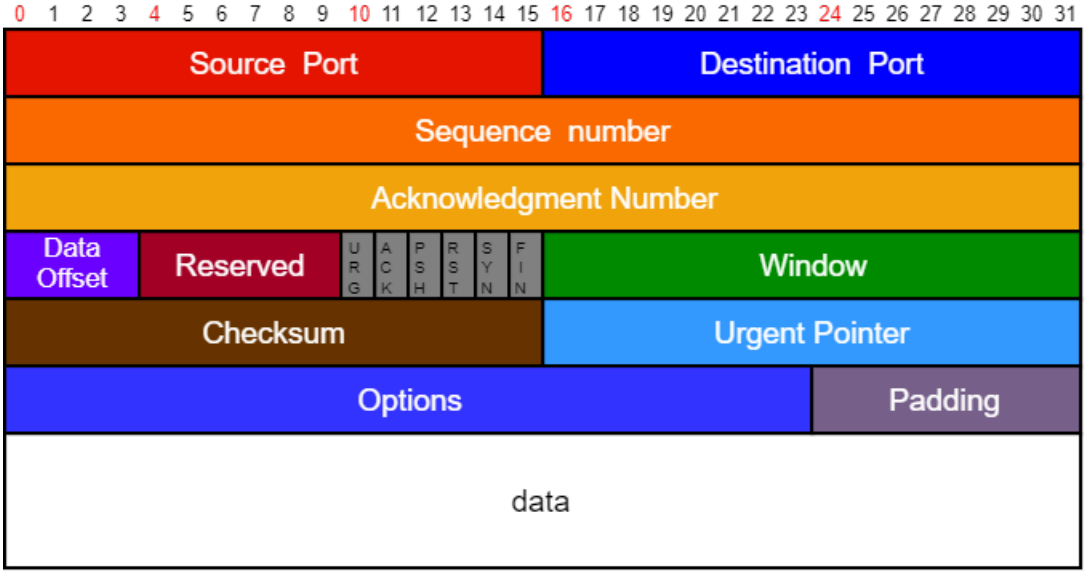


Figure 8.3: TCP packet format.

- **Source Port** (16 bits)
  The source port number
- **Destination Port** (16 bits)
  The destination port number
- **Sequence Number** (32 bits)
  The sequence number of the first data octet in this segment (except when SYN is present). If SYN is present the sequence number is the initial sequence number (ISN) and the first data octet is ISN+1.
- **Acknowledgment Number** (32 bits)
  If the ACK control bit is set this field contains the value of the next sequence number the sender of the segment is expecting to receive. Once a connection is established this is always sent.
- **Data Offset** (4 bits)
  The number of 32 bit words in the TCP Header. This indicates where the data begins. The TCP header (even one including options) is an integral number of 32 bits long.

- **Reserved** (6 bits)
  Reserved for future use. Must be zero.
- **Control Bits** (6 bits (from left to right))

Bit	Meaning
URG	Urgent Pointer field significant
ACK	Acknowledgment field significant
PSH	Push Function
RST	Reset the connection
SYN	Synchronize sequence numbers
FIN	No more data from sender

- **Window** (6 bits)
  The number of data octets beginning with the one indicated in the acknowledgment field which the sender of this segment is willing to accept.
- **Checksum** (16 bits)
  The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header and text. If a segment contains an odd number of header and text octets to be checksummed, the last octet is padded on the right with zeros to form a 16 bit word for checksum purposes.
  The pad is not transmitted as part of the segment. While computing the checksum, the checksum field itself is replaced with zeros. The checksum also covers a 96 bit pseudo header conceptually prefixed to the TCP header.
  This pseudo header contains the Source Address, the Destination Address, the Protocol, and TCP length. This gives the TCP protection against misrouted segments. This information is carried in the Internet Protocol and is transferred across the TCP/Network interface in the arguments or results of calls by the TCP on the IP.

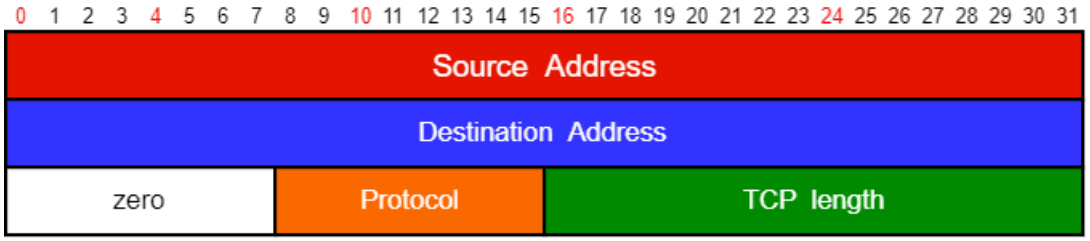


Figure 8.4: Pseudo header.

The TCP Length is the TCP header length plus the data length in octets (this is not an explicitly transmitted quantity, but is computed), and it does not count the 12 octets of the pseudo header.

- **Urgent Pointer** (16 bits)
  This field communicates the current value of the urgent pointer as a positive offset from the sequence number in this segment. The urgent pointer points to the sequence number of the octet following the urgent data. This field is only be interpreted in segments with the URG control bit set.
- **Options** (variable length)
  Options may occupy space at the end of the TCP header and are a multiple of 8 bits in length. All options are included in the checksum. An option may begin on any octet boundary. There are two cases for the format of an option:
  - **Case 1:**
    A single octet of option-kind.

- **Case 2:**
  An octet of option-kind, an octet of option-length, and the actual option-data octets.

The option-length counts the two octets of option-kind and option-length as well as the option-data octets.
Note that the list of options may be shorter than the data offset field might imply.
The content of the header beyond the End-of-Option option must be header padding (i.e., zero). A TCP must implement all options.
Currently defined options include (kind indicated in octal):

Kind	**Length**	**Meaning**
0	-	End of option list
1	-	No-Operation
2	*4*	Maximum Segment Size

- **Padding** (variable length)
  The TCP header padding is used to ensure that the TCP header ends and data begins on a 32 bit boundary. The padding is composed of zeros.

8.2.2 Connection state diagram

A connection progresses through a series of states during its lifetime, that are (Figure 8.6):

- **LISTEN**
  waiting for a connection request from any remote TCP and port.

- **SYN-SENT**
  waiting for a matching connection request after having sent a connection request.

- **SYN-RECEIVED**
  waiting for a confirming connection request acknowledgment after having both received and sent a connection request.

- **ESTABLISHED**
  an open connection in which data received can be delivered to the user. The normal state for the data transfer phase of the connection.

- **FIN-WAIT-1**
  waiting for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request previously sent.

- **FIN-WAIT-2**
  waiting for a connection termination request from the remote TCP.

- **CLOSE-WAIT**
  waiting for a connection termination request from the local user.

- **CLOSING**
  waiting for a connection termination request acknowledgment from the remote TCP.

- **LAST-ACK**
  waiting for an acknowledgment of the connection termination request previously sent to the remote TCP (which includes an acknowledgment of its connection termination request).

- **TIME-WAIT**
  waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request.

- **CLOSED**
  fictional state that represents no connection state at all.

In connection state diagram, each transition shows the event that generates the transition and the operation done as response to the event (Figure 8.5). Hence the event can be seen like the event for which a callback will be called and the operation is the set of instructions implemented in the code of the callback. The response x indicates that no action is performed.

Figure 8.5: Example of transition.

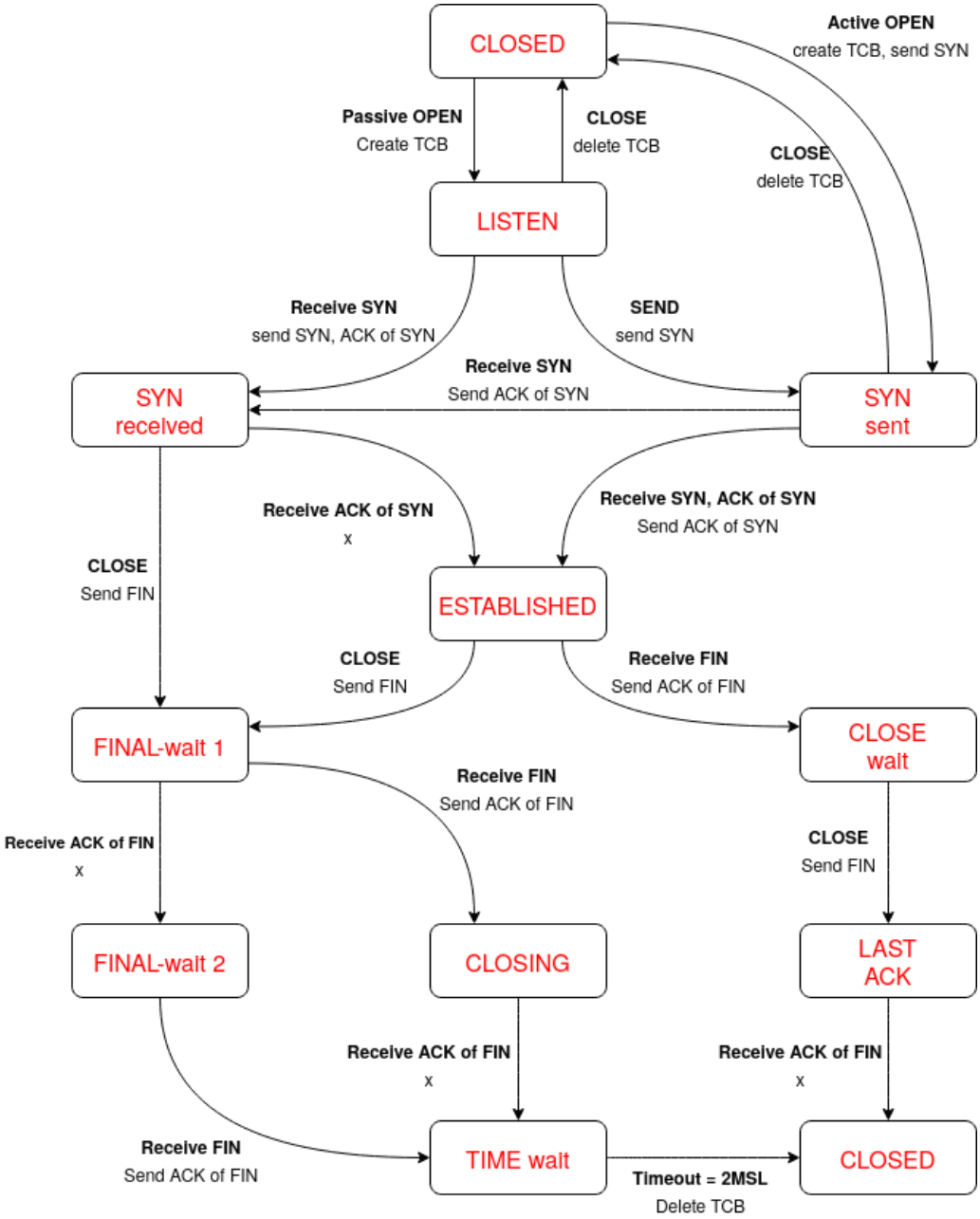


Figure 8.6: Connection state diagram.

8.2.3 Management packet loss

The warranty of the delivery of a packet was implemented at lower level. At layer 4, to understand if the packet is arrived to the receiver, the sender receives a packet called Acknowledgment (ACK).
When the sender sends a packet, he waits for a while. During this period, the sender is almost sure that ACK has to arrive. If it doesn't receive the ACK in this period, it sends again the same packet to the receiver. This behaviour is essential in implementation of sender code to receive a loss (Figure 8.7).

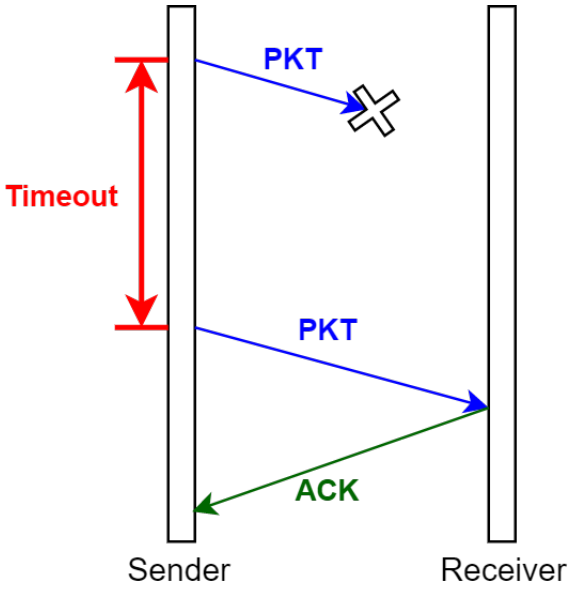


Figure 8.7: Timeout for waiting time of ACK.

If a loss of the ack occurs, the receiver must be able to handle the duplicate packet. Hence the packets need to have an indentifier that allows the receiver to be aware that packet is the same of the first one (Figure 8.8).

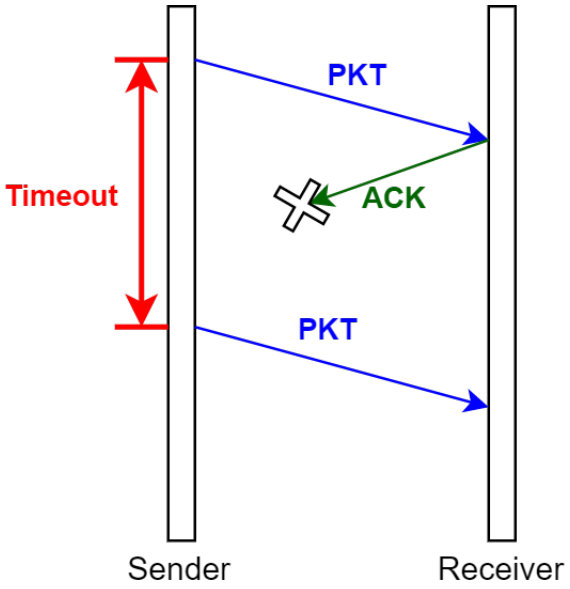


Figure 8.8: Management by receiver of doubled packets.

If the ACKs arrive with a certain delay, we need to enumerate them. The reason can be found looking to Figure 8.9. If the sender sends a packet **PKT 1** and waits for its ACK for a timeout w. If the corresponding ACK arrives after w seconds, the sender has already resent **PKT 1** thinking that it's been lost.
Then suppose that the sender receives **ACK of first PKT 1**, so it sends the next packet **PKT 2** but this will be lost. After a while the sender receives the **ACK of second PKT 1** but, if ACKs are not identified by numbers, the sender can think that the ACK is relative to **PKT 2** because it already receives **ACK of PKT 1**.

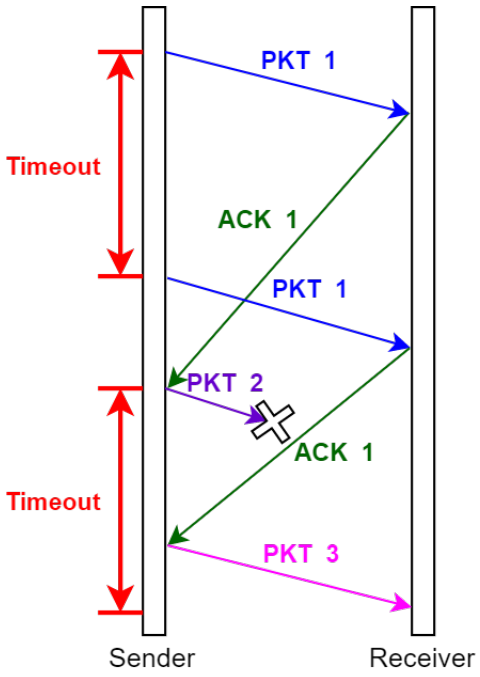


Figure 8.9: Problem with delayed ACK.

During the latency, sending a packet and waiting for its ACK before sending the new one causes waste of time and bandwidth capacibity (Figure 8.10.

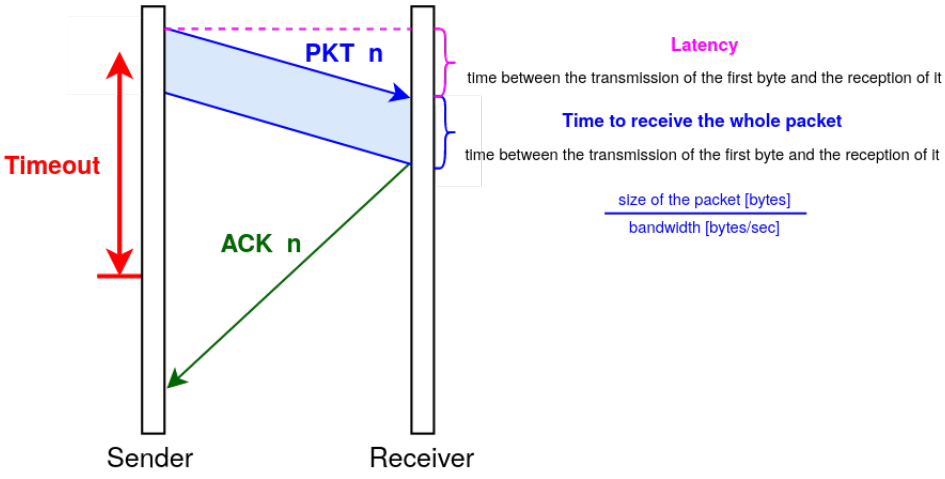


Figure 8.10: Transmission of a packet.

We send in optimistic way more packets to fit the network capacity (pipeline), betting that the whole packet will arrive to destination. The latency becomes negletable with respect to the time needed to send all the packets.

8.2.4 Segmentation of the stream

The buffer is split into segments of bytes and the numbers identify the byte positions. The identifier of the packet is the offset of the stream.
The **sequence number** is the position (offset of the first byte in the segment). The ACK number is the first empty (not yet received) position in the stream (e.g. in Figure 8.11 if segment 1, segment 2 and segment 4 have

been received all the ACK number is 21).

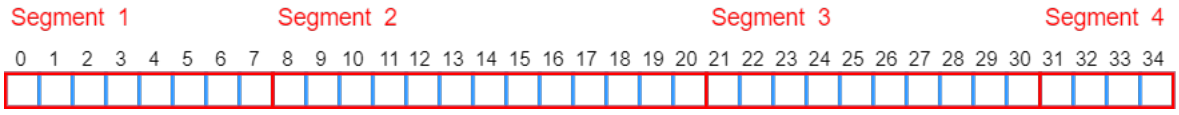


Figure 8.11: Example of segmentation of the stream.

8.2.5 Automatic Repeat-reQuest (ARQ)

ARQ is a control strategy of the errors that detects an error (without correction). Corrupted packets are discarded and there is the request of their retransmissions.

8.2.6 TCP window

A variable window size is usually used and it's increased when there is no packet loss. Variable timeouts are used also in this system. If a packet is lost, the ACK is stopped because it's cumulative and the size of the window is set again to 1.
There are two types of control:

- **Flow control**
  made by receiver
- **Congestion control**
  managing packet losses

TX BUF	RX BUF	NAME	packet id	Timeout	Sender	Receiver actions	Ack type
1	1	Stop & Wait	1-bit	single	Send a packet awaits reply with the same id, after timeout send packet id	Respond ACK with packet id.	single ACK
N	1	go-back-N	log Nbit	window	Send N packets after start ptr. Awaits reply with id. ptr = id	Replies ACK with id only if id is old id + 1	cumulative ack
N	N	Selective Repeat	log Nbit	Single for each frame	Send N packets after the last ACK. Each ACK is specific to each packet, each packet has its own timeout. The sliding window proceeds from the most recent packet received without previous "holes".	ACK replies to each packet with the id of the packet falling in the receiving window.	selective ack
N	N	Sliding window (TCP)	log Nbit	window	Send N packets after start ptr. Each window has its timeout if it is not set (as soon as it is updated) it is set from the first sending. The sliding window resets to the cumulative attack.	Answers ACK cumulative	ACK
N	N	Sliding window (TCP) + SACK	log Nbit	Single for each frame	Send N packets after ptr start. Each ack is cumulative. Each packet has its own timeout. The sliding window resets itself to the cumulative packet	Responds to ACK + Contiguous data blocks	cumulative ACK + SACK

There is usually a threshold, called **ssthresh**, and the actual window size is called **cwnd**. If **cwnd** is less than **ssthresh** the window size will be doubled at next step.
After the first increase of the window size up to the double of **ssthresh**, the window size will increase linearly in the range of $[\texttt{ssthresh}, 2 * \texttt{ssthresh}]$

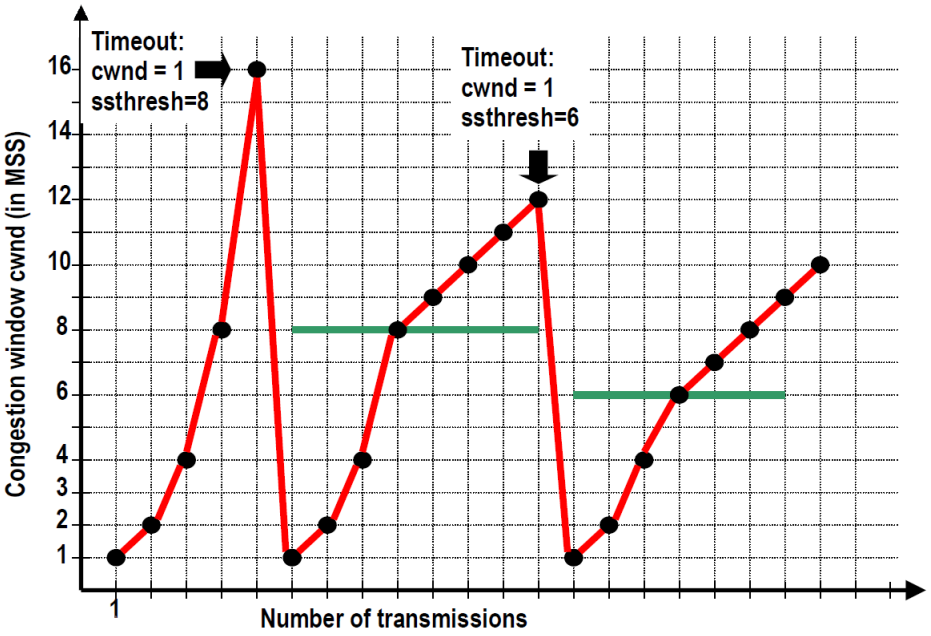


Figure 8.12: Example of window size update.

Chapter 9

HTTP protocol

HTTP protocol was described for the first time in the RFC1945 [?].
The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the simplicity and the speed needed for distributed, collaborative, hypermedia information systems. It is a generic, stateless, object-oriented protocol which can be used for many tasks, such as name servers and distributed object management systems, through extension of its request methods (commands).
It's not the first Hypertext protocol in history because before it there was Hypertalk, made by Apple.
A feature of HTTP is typing the data representation, allowing systems to be built independently w.r.t data being transferred. HTTP has been in use by the World-Wide Web global information initiative since 1990.

9.1 Terminology

- **connection**
  a transport layer virtual circuit established between two application programs for the purpose of communication.

- **message**
  the basic unit of HTTP communication, consisting of a structured sequence of octets matching the syntax defined in Section 4 and transmitted via the connection.

- **request**
  an HTTP request message.

- **response**
  an HTTP response message.

- **resource**
  a network data object or service which can be identified by a URI.

- **entity**
  a particular representation or rendition of a data resource, or reply from a service resource, that may be enclosed within a request or response message. An entity consists of metainformation in the form of entity headers and content in the form of an entity body.

- **client**
  an application program that establishes connections for the purpose of sending requests.

- **user agent**
  the client which initiates a request. These are often browsers, editors, spiders (web-traversing robots), or other end user tools.

- **server**
  an application program that accepts connections in order to service requests by sending back responses.

- **origin server**
  the server on which a given resource resides or is to be created.

- **proxy**
  an intermediary program which acts as both a server and a client for the purpose of making requests on behalf of other clients. Requests are serviced internally or by passing them, with possible translation, on to other servers. A proxy must interpret and, if necessary, rewrite a request message before forwarding it. Proxies are often used as client-side portals through network firewalls and as helper applications for handling requests via protocols not implemented by the user agent.

- **gateway**
  a server which acts as an intermediary for some other server. Unlike a proxy, a gateway receives requests as if it were the origin server for the requested resource; the requesting client may not be aware that it is communicating with a gateway.
  Gateways are often used as server-side portals through network firewalls and as protocol translators for access to resources stored on non-HTTP systems.

- **tunnel**
  a tunnel is an intermediary program which is acting as a blind relay between two connections. Once active, a tunnel is not considered a party to the HTTP communication, though the tunnel may have been initiated by an HTTP request. The tunnel ceases to exist when both ends of the relayed connections are closed.
  Tunnels are used when a portal is necessary and the intermediary cannot, or should not, interpret the relayed communication.

- **cache**
  a program's local store of response messages and the subsystem that controls its message storage, retrieval, and deletion. A cache stores cachable responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests. Any client or server may include a cache, though a cache cannot be used by a server while it is acting as a tunnel.

Any given program may be capable of being both a client and a server; our use of these terms refers only to the role being performed by the program for a particular connection, rather than to the program's capabilities in general. Likewise, any server may act as an origin server, proxy, gateway, or tunnel, switching behavior based on the nature of each request.

9.2 Basic rules

The following rules are used throughout are used to describe the grammar used in the RFC 1945.

OCTET =	<any 8-bit sequence of data>
CHAR =	<any US-ASCII character (octets 0 - 127)>
UPALPHA =	<any US-ASCII uppercase letter "A.."Z">
LOALPHA =	<any US-ASCII lowercase letter "a.."z">
ALPHA =	UPALPHA | LOALPHA
DIGIT =	<any US-ASCII digit "0.."9">
CTL =	<any US-ASCII control character (octets 0 - 31) and DEL (127)>
CR =	<US-ASCII CR, carriage return (13)>
LF =	<US-ASCII LF, linefeed (10)>
SP =	<US-ASCII SP, space (32)>
HT =	<US-ASCII HT, horizontal-tab (9)>
<"> =	<US-ASCII double-quote mark (34)>

9.3 Messages

9.3.1 Different versions of HTTP protocol

- **HTTP/0.9 Messages**
  Simple-Request and Simple-Response don't allow the use of any header information and are limited to a single request method (GET).
  Use of the Simple-Request format is discouraged because it prevents the server from identifying the media type of the returned entity.

```
HTTP-message = Simple-Request | Simple-Response
```

```
Simple-Request  = "GET" SP Request-URI CRLF

Simple-Response = [ Entity-Body ]
```

- **HTTP/1.0 Messages**
  Full-Request and Full-Response use the generic message format of RFC 822 for transferring entities. Both messages may include optional header fields (also known as "headers") and an entity body. The entity body is separated from the headers by a null line (i.e., a line with nothing preceding the CRLF).

```
HTTP-message = Full-Request | Full-Response
```

```
Full-Request = Request-Line
               *(General-Header | Request-Header | Entity-Header)
               CRLF
               [Entity-Body]

Full-Response = Status-Line
                *(General-Header | Request-Header | Entity-Header)
                CRLF
                [Entity-Body]
```

9.3.2 Headers

The order in which header fields are received is not significant. However, it is "good practice" to send General-Header fields first, followed by Request-Header or Response-Header fields prior to the Entity-Header fields.
Multiple HTTP-header fields with the same field-name may be present in a message if and only if the entire field-value for that header field is defined as a comma-separated list.

```
HTTP-header = field-name ":" [ field-value ] CRLF
```

9.3.3 Request-Line

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF

Method          = "GET" | "HEAD" | "POST" | extension-method

extension-method = token
```

The list of methods acceptable by a specific resource can change dynamically; the client is notified through the return code of the response if a method is not allowed on a resource.
Servers should return the status code 501 (not implemented) if the method is unrecognized or not implemented.

9.3.4 Request-URI

The Request-URI is a Uniform Resource Identifier and identifies the resource upon which to apply the request.

```
Request-URI = absoluteURI | abs_path
```

The absoluteURI form is only allowed when the request is being made to a proxy. The proxy is requested to forward the request and return the response. If the request is GET or HEAD and a prior response is cached, the proxy may use the cached message if it passes any restrictions in the Expires header field.
Note that the proxy may forward the request on to another proxy or directly to the server specified by the absoluteURI. In order to avoid request loops, a proxy must be able to recognize all of its server names, including any aliases, local variations, and the numeric IP address.

The most common form of Request-URI is that used to identify a resource on an origin server or gateway. In this case, only the absolute path of the URI is transmitted.

9.3.5 Request Header

The request header fields allow the client to pass additional information about the request, and about the client itself, to the server.
These fields act as request modifiers, with semantics equivalent to the parameters on a programming language method (procedure) invocation.

```
Request-Header = Authorization | From | If-Modified-Since | Referer | User-Agent
```

9.3.6 Status line

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

General Status code

1xx: Informational	Not used, but reserved for future use
2xx: Success	The action was successfully received,
	understood, and accepted.
3xx: Redirection	Further action must be taken in order to complete the request
4xx: Client Error	The request contains bad syntax or cannot be fulfilled
5xx: Server Error	The server failed to fulfill an apparently valid request

Known service code

200	OK
201	Created
202	Accepted
204	No Content
301	Moved Permanently
302	Moved Temporarily
304	Not Modified
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable

9.4 HTTP 1.0

The protocol has no mandatory headers to be added in the request field. This protocol is compliant with HTTP 0.9. To keep the connection alive, "Connection" header with "keep-alive" as header field must be added to request message. The server, receiving the request, replies with a message with the same header value for "Connection".
This is used to prevent the closure of the connection, so if the client needs to send another request, he can use the same connection. This is usually used to send many files and not only one.
The connection is kept alive until either the client or the server decides that the connection is over and one of them drops the connection. If the client doesn't send new requests to the server, the second one usually drops the connection after a couple of minutes.
The client could read the response of request, with activated keep alive option, reading only header and looking to "Content-length" header field value to understand the length of the message body. This header is added only if a request with keep-alive option is done.
This must be done because we can't look only to empty system stream, because it could be that was send only the response of the first request or a part of the response.
Otherwise, when the option keep alive is not used, the client must fix a max number of characters to read from the specific response to his request, because he doesn't know how many character compose the message body.
If you make many requests to server without keep-alive option, the server will reply requests, after the first, with only headers but empty body.

9.4.1 Other headers of HTTP/1.0 and HTTP/1.1

- **Allow**
  lists the set of HTTP methods supported by the resource identified by the Request-URI

- **Accept**
  lists what the client can accept from server. It's important in object oriented typing concept because client application knows what types of data are allowed for its methods or methods of used library

- **Accept-encoding**
  specifies what type of file encoding the client supports (don't confuse it with transfer encoding)

- **Accept-language**
  specifies what language is set by Operating System or it's specified as a preference by client on browser

- **Content-Type**
  indicates the media type of the Entity-Body sent to the recipient. It is often used by server to sppecify which one of the media types, indicated by the client in the Accept request, it will use in the response.

- **Date**
  specifies the date and time at which the message was originated

- **From**
  if given, it should contain an Internet e-mail address for the human user who controls the requesting user agent (it was used in the past)

- **Location**
  defines the exact location of the resource that was identified by the Request-URI (useful for 3xx responses)

- **Pragma**
  It's sent by server to inform that there in no caching systems

- **Referer**
  allows the client to specify, for the server's benefit, the address (URI) of the resource from which the Request-URI was obtained (page from which we clicked on the link). This allows a server to generate lists of back-links to resources for interest, logging, optimized caching, etc. It was added with the born of economy services related to web pages.

- **Server**
  information about the software used by the origin server to handle the request (usually Apache on Unix, GWS(Google Web Server), Azure on Windows, ...)

- **User-agent**
  Version of client browser and Operating System. It's used to:

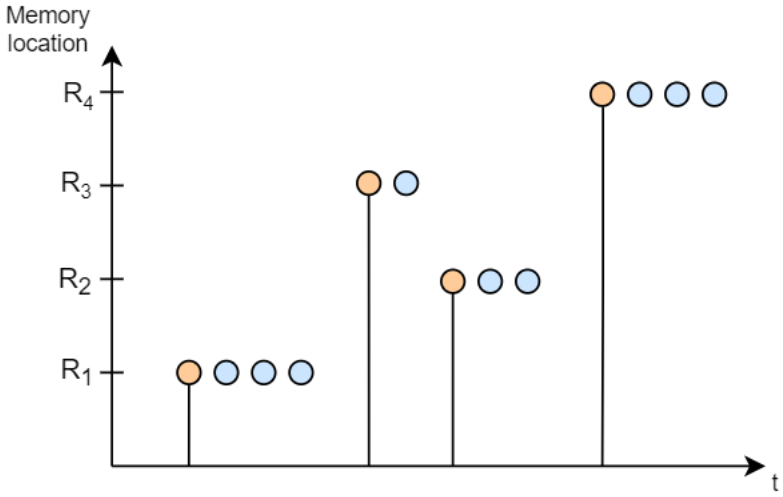
  - adapt responses to application library
  - manage mobile vs desktop web pages

  It's crucial for web applications. If we are the clients and we receive the response from server, we want that the content must change according to the version of browser.
  Infact, there are two different web pages(two different view of the same web page) according to connection by pc and phone, because of different user-agent of these devices. If a mobile phone sends a request to a non-mobile web page, the user agent changes to user agent related to Desktop version.

9.4.2 Caching

It's based on locality principle and was observed on programs execution.

- **Time Locality**
  When a program accesses to an address, there will be an access to it again in the near future with high probability.
  If I put this address in a faster memory (cache), the next access to the same location would be faster.



- **Space Locality**
  If a program accesses to an address in the memory, it's very probable that neighboring addresses would be accessed next.

The caching principle is applied also in Computer Networks, storing of the visited web pages on client system and then updating them through the use of particular headers and requests (see Figure **??**). The purpose of using cache is to reduce traffic over the network and load of the server. The main problem of storing the page in a file, used as a cache, is that the page on the server can me modified and so client's copy can be obsolete.

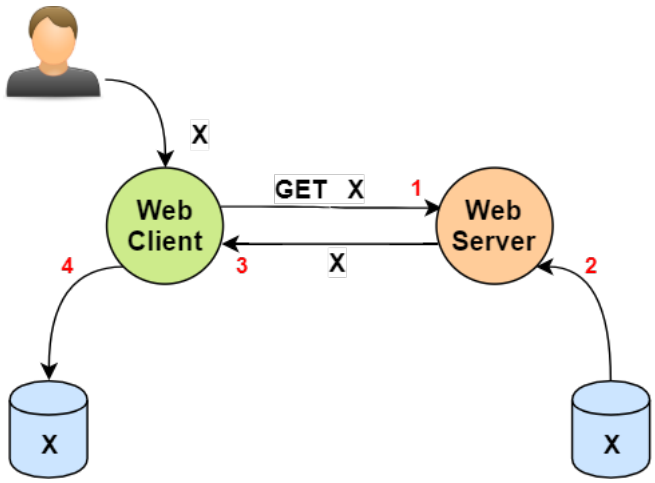


Figure 9.1: First insertion of the resource in the cache.

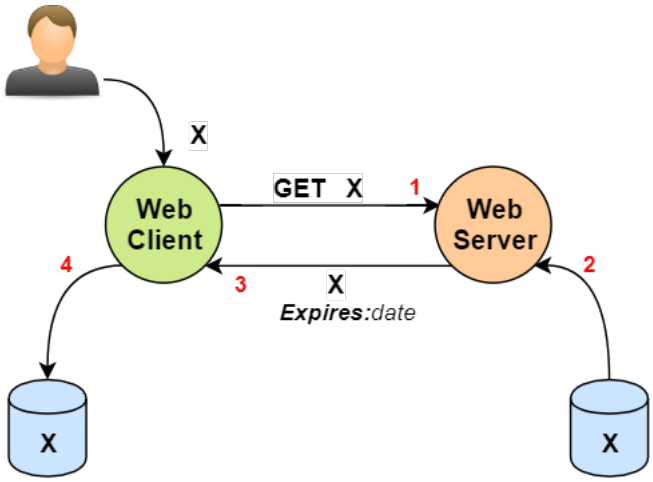
The update of the content of the local cache for the client can happen in three different ways:

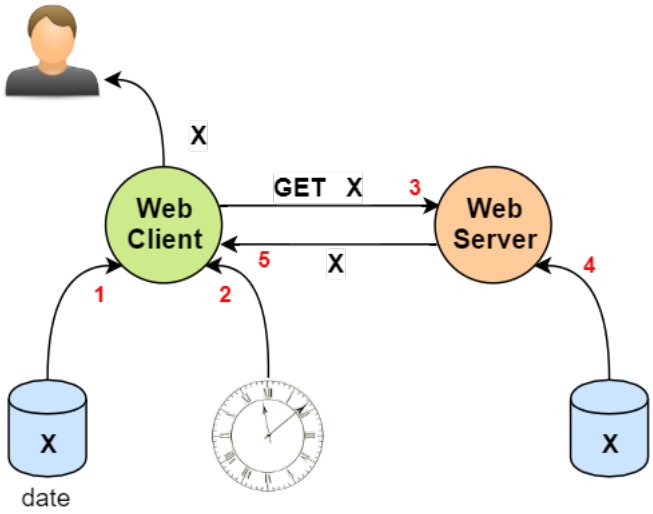
- **Expiration date**

  1. The client asks the resource to the server, that replies with the resource and adding "Expires" header. This is done by the server to specify wan the resource will be considered obsolete.
  2. The client stores a copy of the resource in its local cache.
  3. The client, before sending a new request, checks if it has already the resource he's asking to server. If he has already the resource, he compares the Expiration date, specified by server at phase 1, with the real time clock. A problem of this method is that the server needs to know in advance when the page changes. So the "Expires" value, sent by server, must be:
     - exactly known in advance for periodic changes (E.g. daily paper)

– statistically computed (evaluating the probability of refreshing and knowing a lower bound of duration of resource)

The other problem of this method is that we need to have server and client clocks synchronized. Hence, we need to have date correction and compensation between these systems.

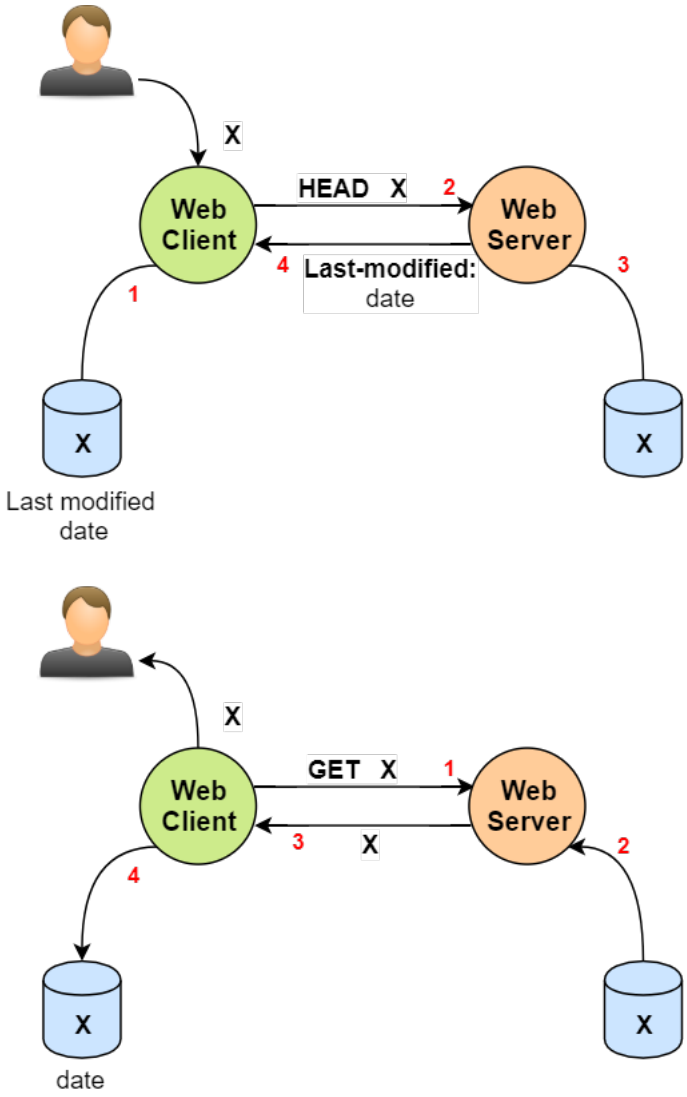




- **Request of only header part**

  1. The client asks the resource to the server as before but now, he stores resource in the cache, within also its "Last-Modified" header value.
  2. The client checks if its copy of the resource is obsolete by making a request to the server of only the header of the resource. This type of request is done by using the *"HEAD"* method.
  3. The client looks to the value of the header "Last-Modified", received by the server. This value is compared with the last-modified header value stored within the resource.
     If the store date was older than new date, the client makes a new request for the resource to the server. Otherwise, he uses the resource in the cache.

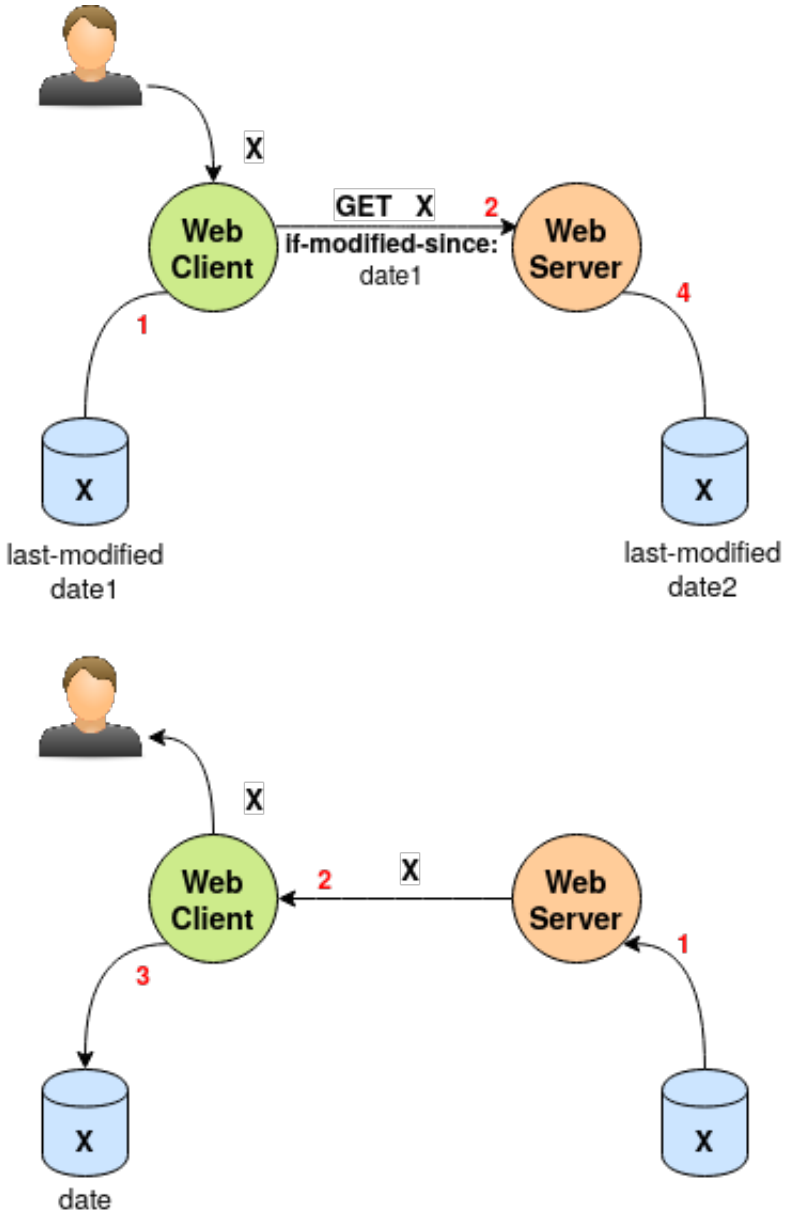
  The problem of this method is that, in the worst case, we send two times the request of the same resource (even if the first one, with "HEAD" method, is less heavy).

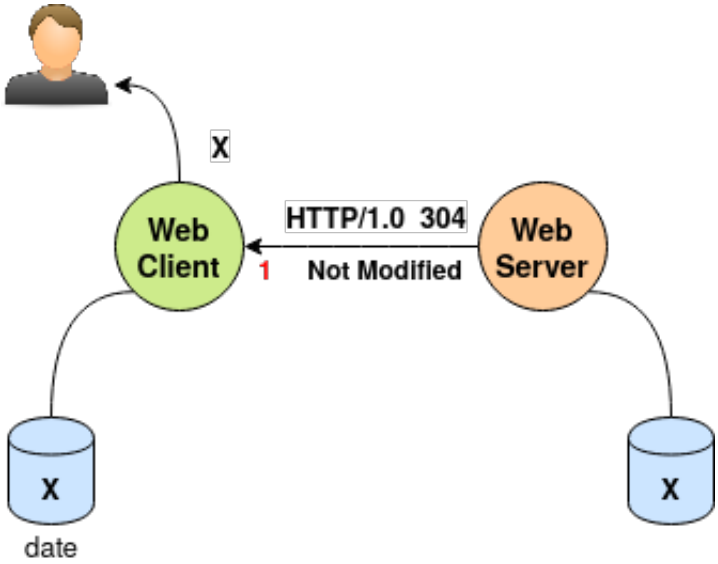


- **Request with if-modified-since header**

1. The client asks the resource to the server as before, storing the resource in the cache within its "Last-Modified" header value.
2. When the client needs again the resource, it sends the request to the server, specifying also "If-Modified-Since" header value as store data.
3. If the server, looking to the resource, sees that its Last-Modified value is more recent than date specified in the request by client, it sends back to the recipient the newer resource.
   Otherwise, it sends to client the message "HTTP/1.0 304 Not Modified".

The positive aspect of this method is that the client can do only a request and obtain the correpte answer without other requests.




X
Web Client
HTTP/1.0 304
Not Modified
1
Web Server
X
X
date

9.4.3 Authorization

1. The client sends the request of the resource to the client
2. The server knows that the resource, to be accessible, needs the client authentication, so it sends the response specifying "WWW-Authenticate:" header, as the following:

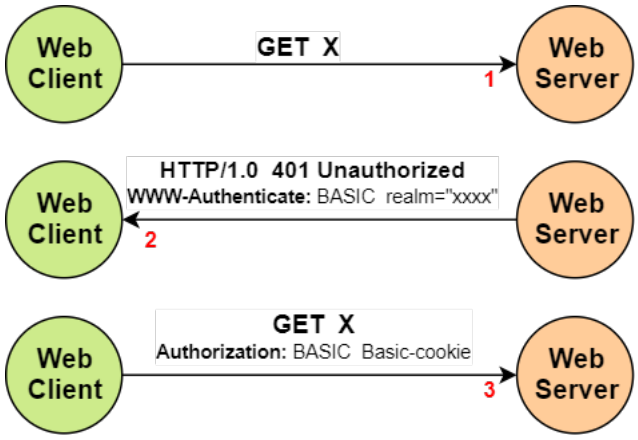
```
WWW-Authenticate: Auth-Scheme Realm="XXXX"
```

Auth-Scheme Type of encryption adopted
Realm "XXXX" refering to the set of users that can access to the resource

3. The client replies with another request of the same resource but specifying also the "Authorization" header value, as the following:

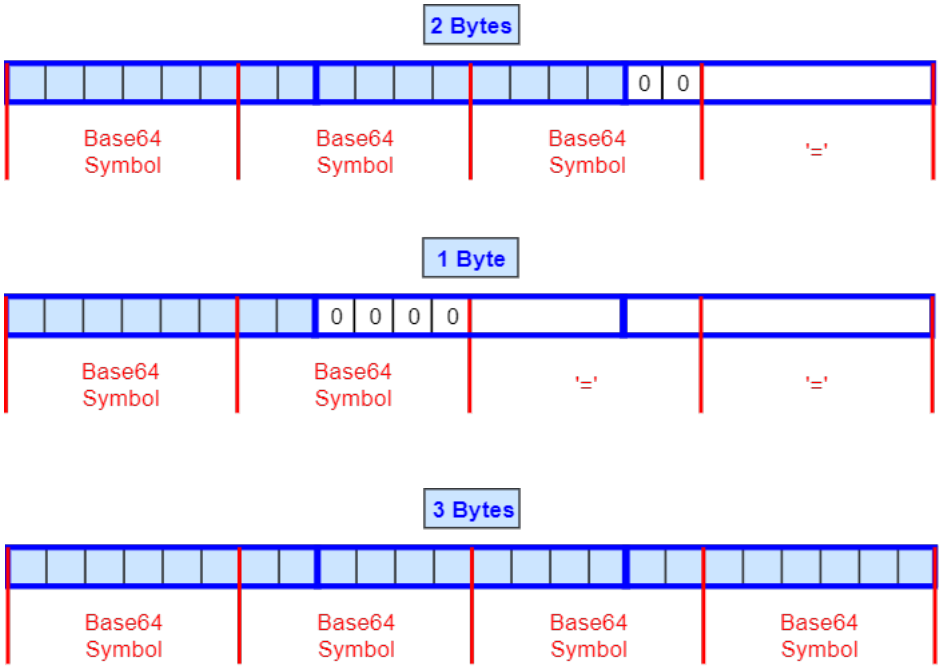
```
WWW-Authenticate: Auth-Scheme Basic-cookie
```

Auth-Scheme Type of encryption adopted
Basic-cookie Base64 encrypted message of the needed for the authentication
(in general basic-ccokie doesn't contain password inside it, it happens only in this case)



9.4.3.1 base64

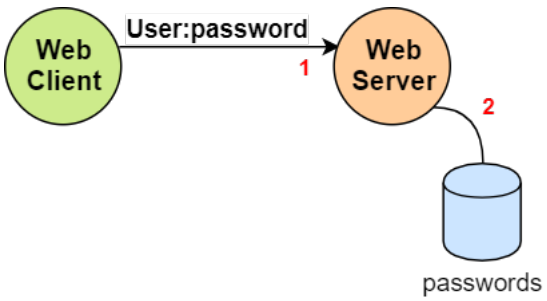
It is very useful for a lot of protocol like HTTP, that doesn't support format different than text of characters. For example with SMTP, all the mail contents must be text, hence images or other binary files are encrypted with base64.
Starting from a stream of bytes, we are going to convert numbers, described by each byte, into ANSI character symbols. These selected symbols, from the Table **??** of all the 64 symbols, are generated looking the values of subsequences of 6 bits.
If the stream of bytes is not composed by a multiple of 24 bits, base64 pad whole missing bytes with symbol '=' (not defined as one of the 64 symbols of the alphabet) and other single missing bits with 0 values.



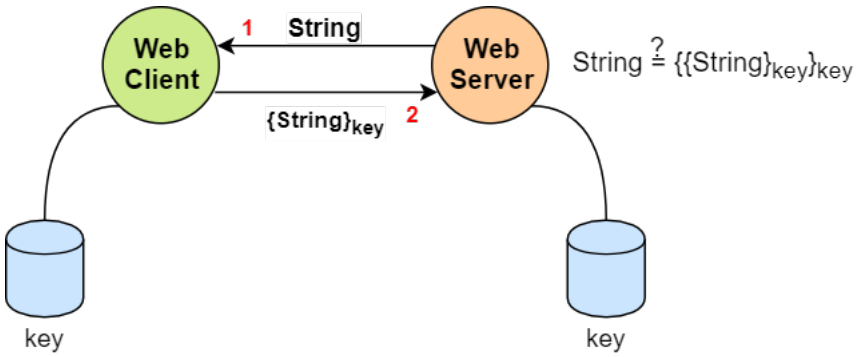
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/
pad =							

9.4.3.2 Auth-schemes

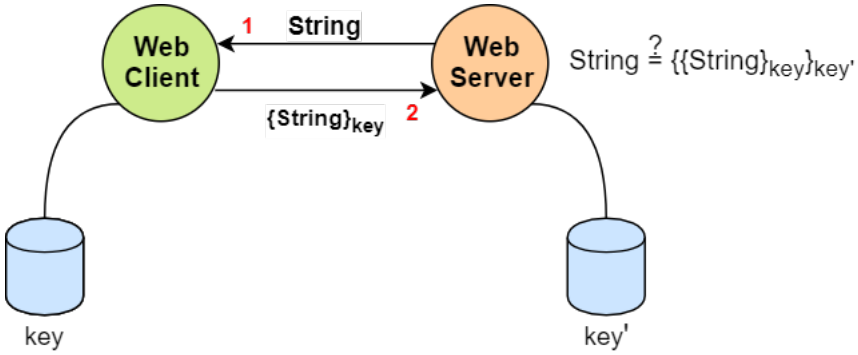
- **BASIC**



- **Challenge** (symmetric version)



- **Challenge** (asymmetric version)



9.5 HTTP 1.1

The architecture of the model is in RFC2616 [?]. It has by default the option keep alive actived by default with respect to HTTP 1.0. It has the mandatory header "Host" followed by the hostname of the remote system, to which the request or the response is sent. The headers used in HTTP/1.0 are used also in HTTP/1.1, but in this new protocol there are new headers not used in the previous one. The body is organized in chunks, so we need the connection kept alive to manage future new chunks.
This is useful with dynamic pages, in which the server doesn't know the length of the stream in advance and can update the content of the stream during the extablished connection, sending a fixed amount of bytes to client. We can check if the connection is chunked oriented, looking for the header "Transfer-Encoding" with value "chunked".
Each connection is composed by many chunks and each of them is composed by chunk length followed by chunk body, except for the last one that has length 0 (see Figure 9.2). The following grammar represents how the body is organized:

```
Chunked-Body    = *chunk
                  last-chunk
                  trailer
                  CRLF

chunk           = chunk-size [ chunk-extension ] CRLF
                  chunk-data CRLF

chunk-size      = 1*HEX
last-chunk      = 1*("0") [ chunk-extension ] CRLF

chunk-extension= *( ";" chunk-ext-name [ "=" chunk-ext-val ] )

chunk-ext-name = token
chunk-ext-val  = token | quoted-string
chunk-data     = chunk-size(OCTET)
trailer         = *(entity-header CRLF)
```

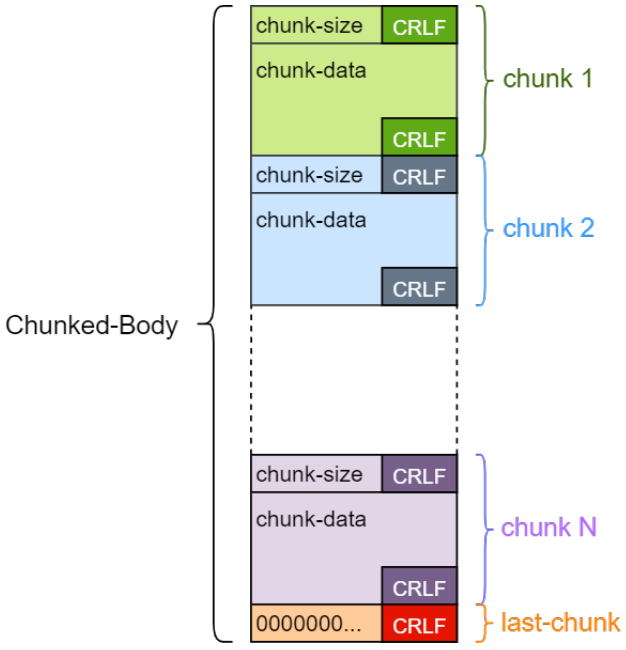


Figure 9.2: Chunked body.

9.5.1 Caching based on HASH

It's like the caching mechanism used looking to "Last-Modified" header value through HEAD request. The organization is as follows:

1. The client asks the resource to the server, he stores resource in the cache, within also its "Etag" header value.
2. The client checks if its copy of the resource is obsolete by making a request to the server of only the header of the resource. This type of request is done by using the *"HEAD"* method.
3. The client looks to the value of the header "Etag", received by the server. This value is compared with the "Etag" header value stored within the resource, because everytime that a file changes, its hash code is computed again.
   If the store date has different hash code from one received, the client makes a new request for the resource to the server. Otherwise, he uses the resource in the cache.

9.5.2 URI

In URI, there is the encapsulation of the operation done in the past, to have a resource from a server [**?**]. The following phases are related to **ftp** application:

1. Open the application ftp
2. Open the server File System, through a general login
3. Select the resource you want to use and download it

```
URI           = ( absoluteURI | relativeURI ) [ "#" fragment ]

absoluteURI   = scheme ":" *( uchar | reserved )
relativeURI   = net_path | abs_path | rel_path

net_path      = "//" net_loc [ abs_path ]
abs_path      = "/" rel_path
rel_path      = [ path ] [ ";" params ] [ "?" query ]

path          = fsegment *( "/" segment )
fsegment      = 1*pchar
segment       = *pchar

params        = param *( ";" param )
param         = *( pchar | "/" )

scheme        = 1*( ALPHA | DIGIT | "+" | "-" | "." )
net_loc       = *( pchar | ";" | "?" )
query         = *( uchar | reserved )
fragment      = *( uchar | reserved )

pchar         = uchar | ":" | "@" | "&" | "=" | "+"
uchar         = unreserved | escape
unreserved    = ALPHA | DIGIT | safe | extra | national

escape        = "%" HEX HEX
reserved      = ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+"
extra         = "!" | "*" | "'" | "(" | ")" | ","
safe          = "$" | "-" | "_" | "."
unsafe        = CTL | SP | <"> | "#" | "%" | "<" | ">"
national      = <any OCTET excluding ALPHA, DIGIT,
                reserved, extra, safe and unsafe>
```

Hence Uniform Resource Identifiers are simply formatted strings which identify via name, location, or any other characteristic a network resource. The following example refers to Relative URI:

```
//net_loc/a/b/c?parameters
```

//net_loc	Server location
/a/b/c	Resource with the path
?parameters	Set of parameters

9.5.3 HTTP URL

It's a particulare instance of absolute URI, with scheme "http".

```
http_URL        = "http:" "//" host [ ":" port ] [ abs_path ]

host            = <A legal Internet host domain name
                   or IP address (in dotted-decimal form),
                   as defined by Section 2.1 of RFC 1123>

port            = *DIGIT
```

There are also other schemes that are not used for web [**?**], for example **ftp** to download resources.

9.6 Dynamic pages

Dynamic pages are created on fly by some web applications in the server. The client makes a request to the server function with some parameters (Figure 9.3).
This approach is based on **Common Gateway Interface (CGI-bin)**, whose name comes from first network applications that were binary. Then the evolution of web applications brings to two types of program:

- **Script Server programs**
  based on PHP, ASP.net

- **Server application (based on Java)** written through J2EE, TomCat and Websphere

The result of these programs are written at Presentation layer, like HTML source. To use the CGI-bin paradigm, the client needs to create a request for a file to be executed and not transfered. For convention, the server usually has its executable files in **"/CGI-bin"** path of the server. The following HTTP URL is the request to the server, made by the client, for the function **f**:

```
http://www.hello.com/CGI-bin/f?a=10&b=20&c=%22ciao%22
```

In this example the client is asking to server **www.hello.com**, using an HTTP URL, the result of the call of function **f**. The symbol **?** defines from which point the parameters of the function are specified. In this case there are three parameters: **a** with value **10**, **b** with value **20** and **c** with value **%22ciao%22**.
There are particular symbols, used in URL:

?	Beginning of parameters section
%	*Escape character* followed by the hex number that defines the symbol you want to code
&	*Separator character* character between each couple of specified parameters

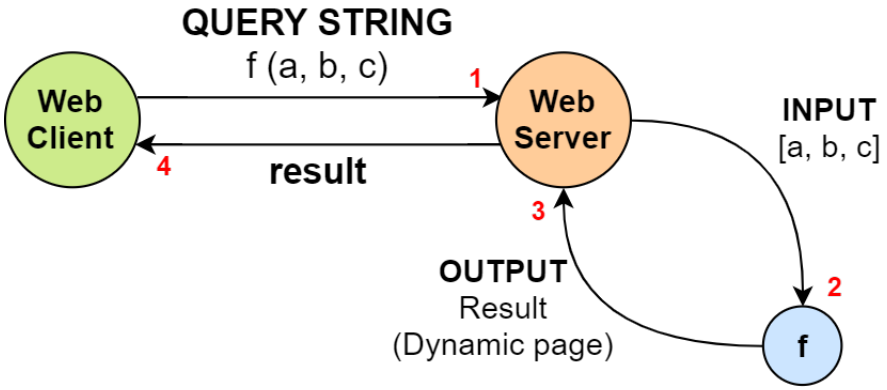


Figure 9.3: Example of CGI application.

9.7 Proxy

The implementation of the proxy depends on the type of protocol used:

- **HTTP**
  If the client wants to use a proxy, doing a *GET* request, he needs to modify its behaviour with the following steps:

  1. **Connection of Client to Web Proxy instead of the server**
     The client needs to change address and port w.r.t. proxy ones, instead of server ones.
  2. **Specify the absolute URI of the requested resource**
     Otherwise proxy doesn't to which one the message needs to be sent. Hence he couldn't forward as it is the request.
     The proxy can analyze the content of data they need to transmit, obtaining the absolute URI and doing another request.

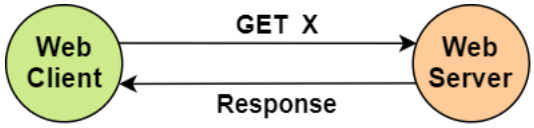


Figure 9.4: Direct access.

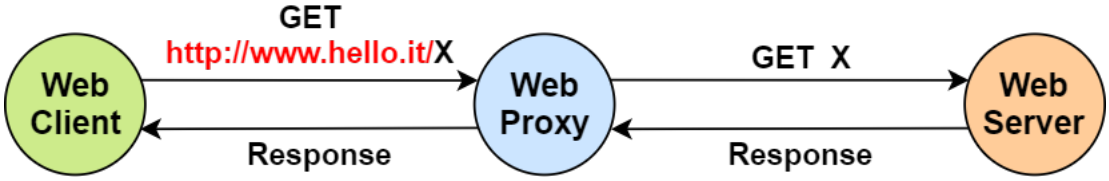


Figure 9.5: Proxy access.

- **HTTPS**
  Data are sent over encrypted channel (TLS) and the proxy can be implemented in two different ways:

  – **Split the encrypted channel**
    The proxy has an encrypted channel with the client and one with the server. This approach can be applied only when we have a trusted proxy (E.g. WAF) because the proxy needs to access data to forward them.

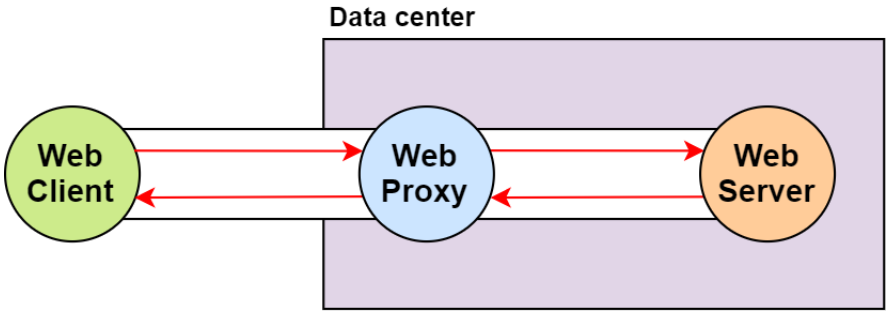


Figure 9.6: Proxy as WAF in HTTPS.

– **Change default behaviour of proxy**
The proxy in this case can only forward encrypted data without knowing anything about them. In this case, proxy works as a Layer-4 gateway and creates a tunnel between client and server [**?**].
In HTTPS the client uses the method *CONNECT* to tell to the proxy to work as a tunnel. The proxy, receiving the *CONNECT* request, extablishes the secure connection between client and server (through the preliminary exchange of keys with Diffie-Hellmann).
The proxy sends HTTP response to the client if the **connect()** call succeded. Then the client can send encrypted data as *raw data* and the proxy will not access them but only forward them. With *CONNECT* request, the client asks to open a connection to web host.
The proxy needs to create two processes (Figure 9.9):

  * **Parent process**
    It reads response from the server and forwards it to the client. When the connection will be closed from the server, it will kill its child process.
  * **Child process**
    It reads request from the client and forwards it to the server.

In a browser, when you type an address or server name, the connection starts by default using HTTP. Then the remote server replies with a HTTP response with redirection to an HTTPS URL.

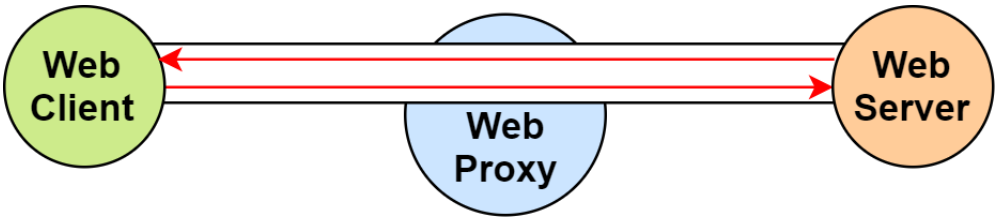


Figure 9.7: Tunneling using proxy in HTTPS.

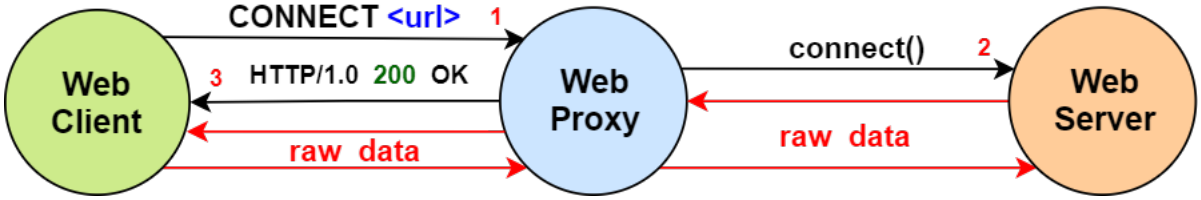


Figure 9.8: CONNECT request in HTTPS.

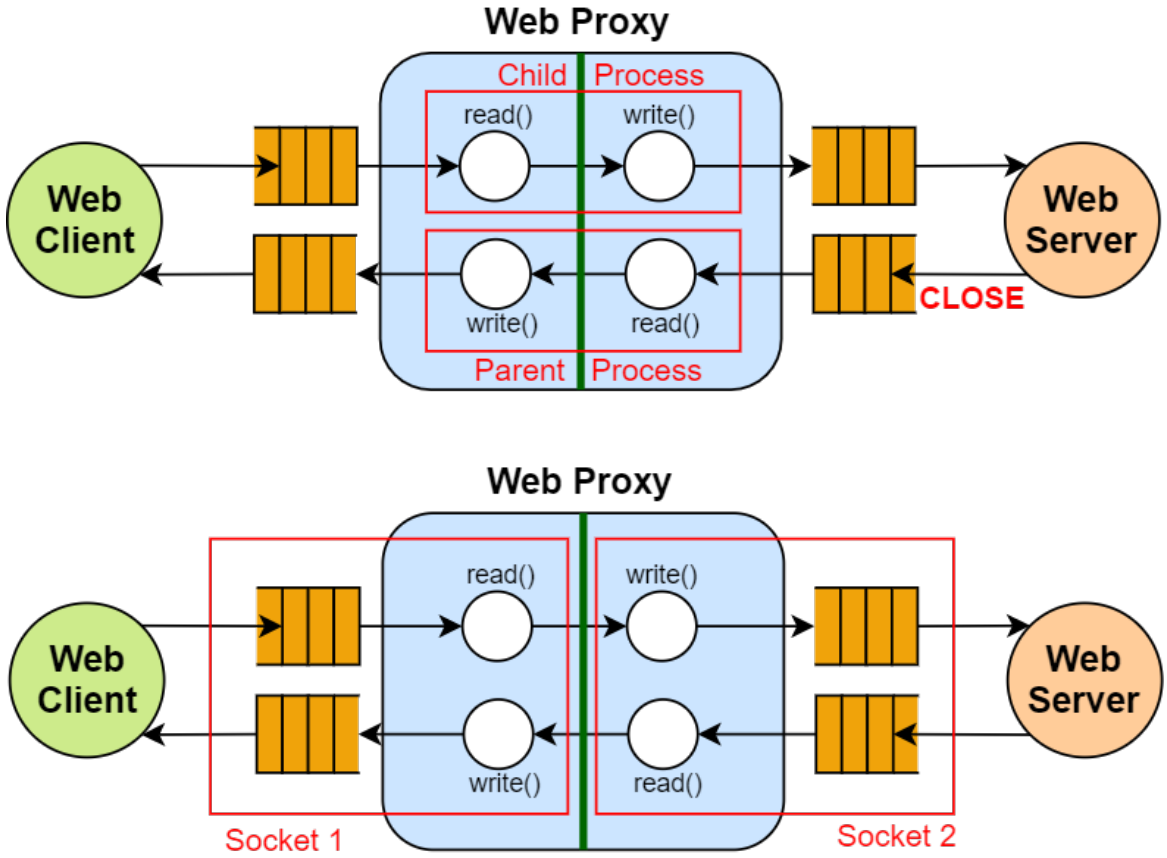


Figure 9.9: How proxy works with HTTPS.

Chapter 10

Resolution of names

The following section will talk about history of technologies under the resolution of server names in URL to their IP addresses, needed to extablish the connection.

10.1 Network Information Center (NIC)

This type of architecture was used in the past to resolve names. Each client has its own file **HOSTS.txt**, with resolution of names. The client shared its file with a central system, called **NIC** (Figure 10.1). This system collects all the files, like an hub, and shared resolution names to other clients.
This architecture is unfeasable and not scalable with nowadays number of IP addresses, because the files become very huge and transfering becomes very slow.

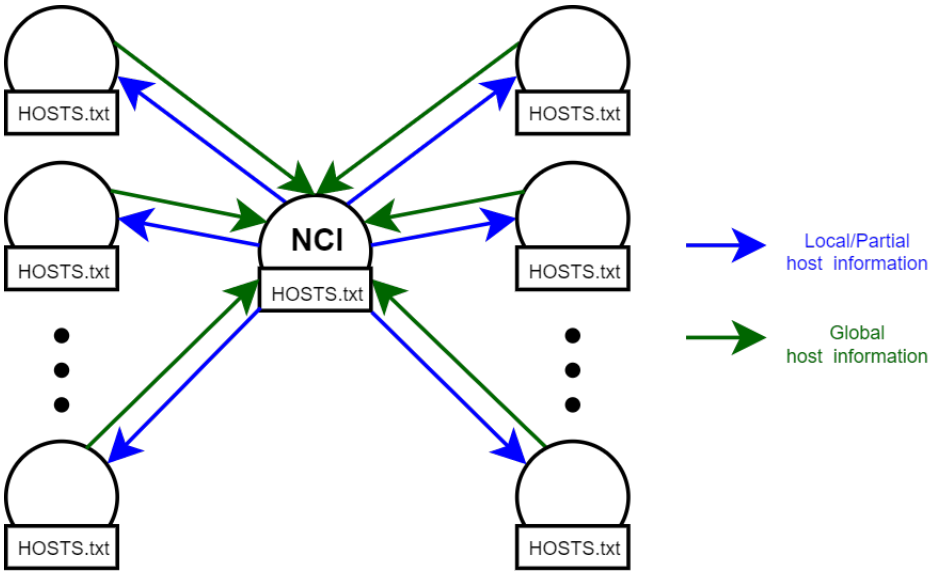


Figure 10.1: How NIC worked.

10.2 Domain Name System (DNS)

The file **HOSTS.txt** is yet used in nowadays UNIX systems (Section 10.1). The specified host name is searched in local /**etc**/**hosts.txt**, that contains local and private addresses resolution table, and if not found, it will be searched through DNS [**?**].

10.2.1 Goals

1. Names should not be required to contain network identifiers, addresses, routes, or similar information as part of the name.

2. The sheer size of the database and frequency of updates suggest that it must be maintained in a distributed manner, with local caching to improve performance.
   Approaches that attempt to collect a consistent copy of the entire database will become more and more expensive and difficult, and hence should be avoided.
   The same principle holds for the structure of the name space, and in particular mechanisms for creating and deleting names; these should also be distributed.

3. Where there are tradeoffs between the cost of acquiring data, the speed of updates, and the accuracy of caches, the source of the data should control the tradeoff.

4. The costs of implementing such a facility dictate that it be generally useful, and not restricted to a single application.
   We should be able to use names to retrieve host addresses, mailbox data, and other as yet undetermined information. All data associated with a name is tagged with a type, and queries can be limited to a single type.

5. Because we want the name space to be useful in dissimilar networks and applications, we provide the ability to use the same name space with different protocol families or management.
   For example, host address formats differ between protocols, though all protocols have the notion of address. The DNS tags all data with a class as well as the type, so that we can allow parallel use of different formats for data of type address.

6. We want name server transactions to be independent of the communications system that carries them.
   Some systems may wish to use datagrams for queries and responses and only establish virtual circuits for transactions that need the reliability (e.g., database updates, long transactions); other systems will use virtual circuits exclusively.

7. The system should be useful across a wide spectrum of host capabilities.
   Both personal computers and large timeshared hosts should be able to use the system, though perhaps in different ways.

10.2.2 Hierarchy structure

Hierarchy permits to manage a lot of nambers of domain names and IP addresses, reducing the time spent to resolve them. Given for example the host name **www.dei.unipd.it**, we have a **Name Server (NS)** for each of the domain name inside it (Figure 10.2). The tree hierarchy has a name server for each one of its internal nodes.The name server gives us only the name of the name server of the lower level to which we need to go.
To obtain the IP address of this name server, we need to ask, to name server of upper layer, a **glue record**. The glue record is an additional information that is needed by us to understand how to reach that name server. Hence the glue record is the IP address of NS of the lower level in hierarchy.
For each request to NS, we obtain also the expiration time information because a caching approach is adopted in DNS but at level 4. There are 13 root name servers that are obtained when asking resolution to root.
In reality root name servers are more than 13 but the communication used in DNS is made through UDP and this type of connection supports only 13 simultaneously transfers. The local DNS server for the device, managed by my network provider, contains the 13 root servers and permits us to reach at least one DNS root server.
The 13 DNS root servers are added locally during the installation of local DNSs and updated assuming that at least one root server of them can be reachable. There is no address record for the root.
In general structure of the queries to name servers, we ask only the resolution for a specific domain that composes the whole name (Figure 10.3).
To use a caching system efficiently, we need to make a recursive query, sending the request of resolution of the whole name with all its domains (Figure 10.4). All the name servers, where the query passes throuh, store information about resolution. This system is never applied as it is.
In reality an hybrid version is implemented, using only partial recursion (Figure 10.5). Local DNS usually has huge cache with main important names and also first and second level have caches. So local DNS rarely asks

resolution to TOP Level Domain or Root.
Recursive query option in dig command is made by a flag, default set to yes and used in UDP packet as an additional information. The Root Name Server decides if it wants to accept recursive query or if not, how many domains can resolve. I can group some domains, defining a zone, so I can use only a name server for a specific zone to solve many domains together (Figure 10.6). So the name servers are authorithative over zones and not only single domains.
The creation of the zones are used to manage easily the responsability of companies and their organization over the zones, grouping domains. Another reason for this partition in zones is the presence of some domains with few names, that it's better to group with other domains.

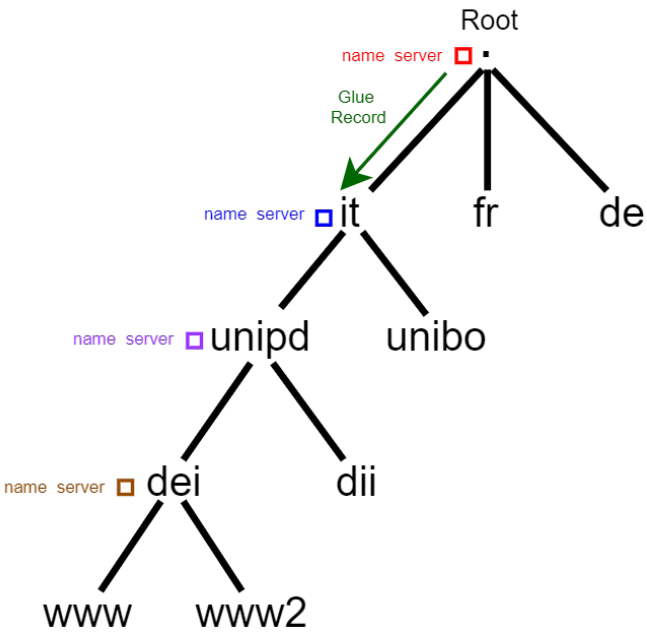


Figure 10.2: DNS structure.

Listing 10.1: Example of default DNS queries using dig.

```
//Ask for root name server to the default name server
dig -t NS -n .

//Ask for address of root name server "a.root-servers.net", previously chosen
dig -t A -n a.root-servers.net

//Ask for "it" name server to the "a.root-servers.net" address, previously chosen
dig @198.41.0.4 -t NS it

//Ask for address of "nameserver.cnr.it" name server,  previously chosen for "it" domain
dig @198.41.0.4 -t A  nameserver.cnr.it

//Ask for "unipd.it" name server to the "nameserver.cnr.it" address
dig @194.119.192.34 -t NS -n unipd.it

//Ask for "unipd.it" name server to the "nameserver.cnr.it" address
dig @194.119.192.34 -t A unipd.it

//Ask for "dei.unipd.it" name server to one ("mail.dei.unipd.it"
dig @147.162.1.100 -t NS  dei.unipd.it

//Ask for address of "mail.dei.unipd.it" name server, previously chosen
dig @147.162.1.2 -t A  mail.dei.unipd.it

//Ask for address of "www.dei.unipd.it" to "mail.dei.unpd.it" name server, previousy chosen
dig @147.162.2.100 -t A  www.dei.unipd.it
```

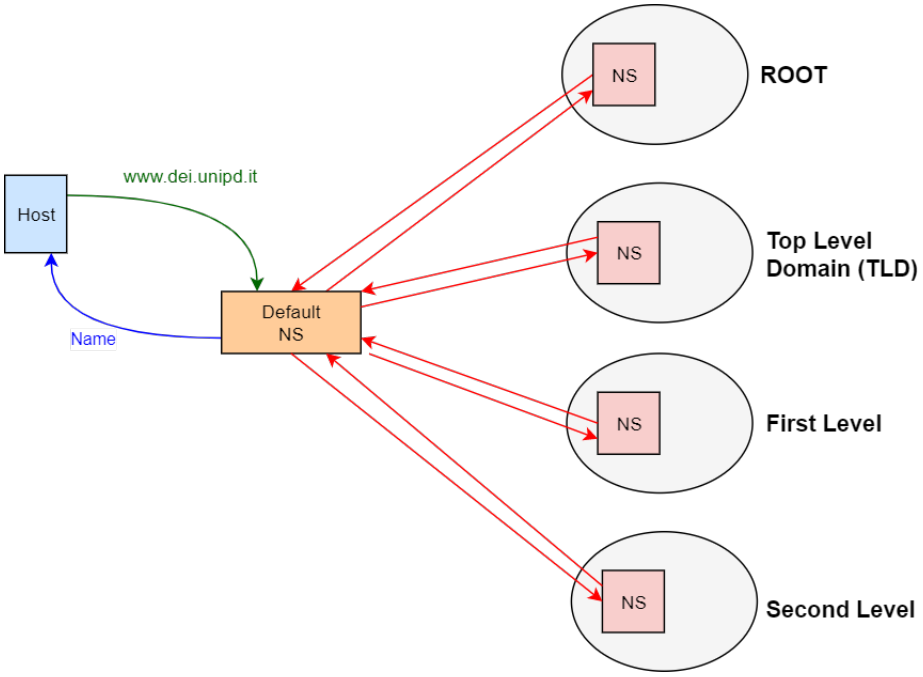


Figure 10.3: Default DNS behaviour without caching.

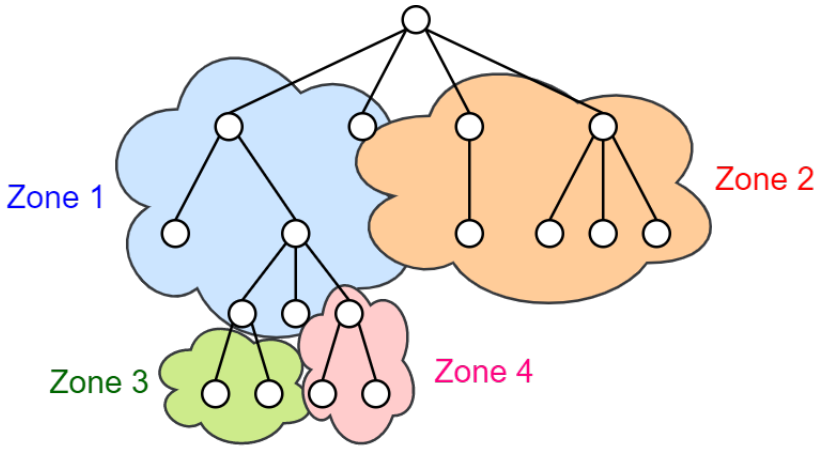


Figure 10.6: Example of partitioning into zones.

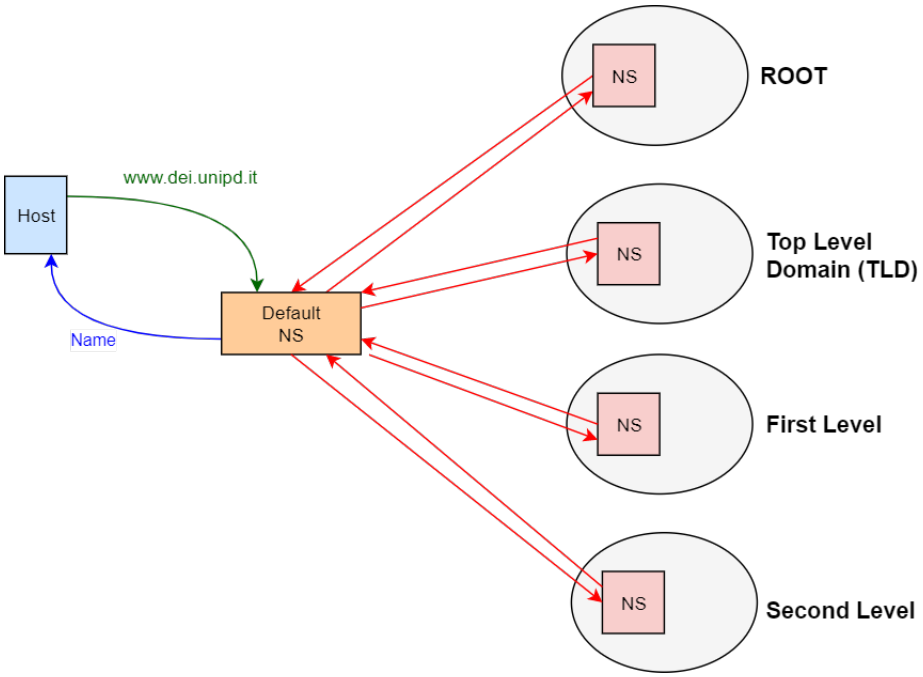


Figure 10.4: Completely recursive DNS structure.

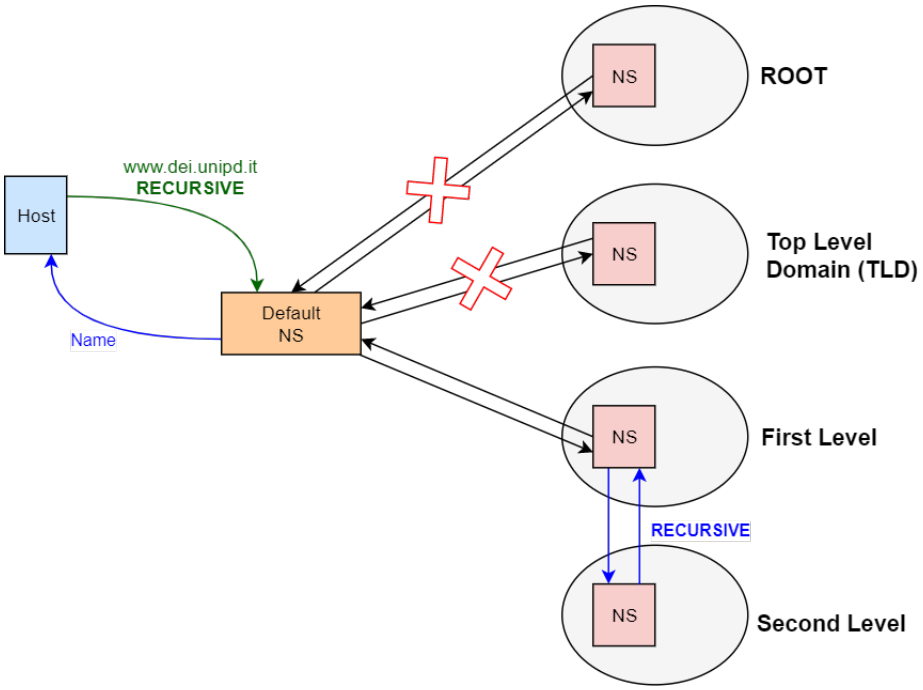


Figure 10.5: Hybrid DNS structure.

Appendix A

Shell

A.1 Commands

man man		Shows info about man command and lists all the sections of the manual.
strace objFile		Lists all the system calls used in the program.
ltrace objFile		Lists all the library calls used in the program.
gcc -o objFile source **-v**		Lists all the path of libraries and headers used in creation of objFile.
netstat	-t	Lists all the active TCP connections showing domain names.
	-u	Lists all the active UDP connections showing domain names.
	-n	Lists all the active, showing IP and port numbers.
nslookup domain		Shows the IP address related to the domain (E.g. IP of www.google.it)
dig @server name type		DNS lookup utility. **server** name or IP address of the name server to query **name** name of the resource record that is to be looked up **type** type of query is required (ANY, A, MX, SIG, etc.) if no type is specified, A is performed by default
wc [file]		Prints in order newlines, words, and bytes (characters) counts for file if file not specified or equal to -, counts from stdin.
route -n		Show numerical addresses instead of trying to determine symbolic hostnames in routing table.
arp -a		List all the MAC addresses stored after some ARP requests and replies made by our ethernet interfaces.

A.2 UNIX Files

/etc/hosts	Local resolution table.
/etc/services	List all the applications with their port and type of protocol (TCP/UDP).
/etc/protocols	Internet protocols.
/usr/include/x86_64-linux-gnu/bits/socket.h	List all the protocol type possible for socket.
/usr/include/x86_64-linux-gnu/sys/socket.h	Definition of struct sockaddr and specific ones.

Appendix B

vim

B.1 .vimrc

In this section there will be shown the file **.vimrc** that can be put in the user home (∼ or **$HOME** or –) or in the path /**usr**/**share**/**vim**/ to change main settings of the program.

Listing B.1: .vimrc

```
syntax on
set number
filetype plugin indent on
set tabstop=4
set shiftwidth=4
set expandtab
set t_Co=256
```

B.2 Shortcuts

Main

Esc	Gets out of the current mode into the "command mode". All keys are bound of commands
i	"Insert mode" for inserting text.
:	"Last-line mode" where Vim expects you to enter a command.

Navigation keys

h	moves the cursor one character to the left.
j or **Ctrl** + **J**	moves the cursor down one line.
k or **Ctrl** + **P**	moves the cursor up one line.
l	moves the cursor one character to the right.
0	moves the cursor to the beginning of the line.
$	moves the cursor to the end of the line.
^	moves the cursor to the first non-empty character of the line
w	move forward one word (next alphanumeric word)
W	move forward one word (delimited by a white space)
5w	move forward five words
b	move backward one word (previous alphanumeric word)

B	move backward one word (delimited by a white space)
5b	move backward five words
G	move to the end of the file
gg	move to the beginning of the file.

Navigate around the document

h	moves the cursor one character to the left.
(	jumps to the previous sentence
)	jumps to the next sentence
{	jumps to the previous paragraph
}	jumps to the next paragraph
[[	jumps to the previous section
]]	jumps to the next section
[]	jump to the end of the previous section
][	jump to the end of the next section

Insert text

h	moves the cursor one character to the left.
a	Insert text after the cursor
A	Insert text at the end of the line
i	Insert text before the cursor
o	Begin a new line below the cursor
O	Begin a new line above the cursor

Special inserts

:r [filename]	Insert the file [filename] below the cursor
:r ![command]	Execute [command] and insert its output below the cursor

Delete text

x	delete character at cursor
dw	delete a word.
d0	delete to the beginning of a line.
d$	delete to the end of a line.
d)	delete to the end of sentence.
dgg	delete to the beginning of the file.
dG	delete to the end of the file.
dd	delete line
3dd	delete three lines

Simple replace text

r}text}	Replace the character under the cursor with {text}
R	Replace characters instead of inserting them

Copy/Paste text

yy	copy current line into storage buffer
["x]yy	Copy the current lines into register x
p	paste storage buffer after current line
P	paste storage buffer before current line
["x]p	paste from register x after current line
["x]P	paste from register x before current line

Undo/Redo operation

u	undo the last operation.
Ctrl+r	redo the last undo.

Search and Replace keys

/search_text	search document for search_text going forward
?search_text	search document for search_text going backward
n	move to the next instance of the result from the search
N	move to the previous instance of the result
:%s/original/replacement	Search for the first occurrence of the string "original" and replace it with "replacement"
:%s/original/replacement/g	Search and replace all occurrences of the string "original" with "replacement"
:%s/original/replacement/gc	Search for all occurrences of the string "original" but ask for confirmation before replacing them with "replacement"

Bookmarks

m {a-z A-Z}	Set bookmark {a-z A-Z} at the current cursor position
:marks	List all bookmarks
'{a-z A-Z}	Jumps to the bookmark {a-z A-Z}

Select text

v	Enter visual mode per character
V	Enter visual mode per line
Esc	Exit visual mode

Modify selected text

	Switch case
d	delete a word.
c	change
y	yank
>	shift right
<	shift left
!	filter through an external command

Save and quit

:q	Quits Vim but fails when file has been changed
:w	Save the file
:w new_name	Save the file with the new_name filename
:wq	Save the file and quit Vim.
:q!	Quit Vim without saving the changes to the file.
ZZ	Write file, if modified, and quit Vim
ZQ	Same as :q! Quits Vim without writing changes

B.3 Multiple files

- **Opening many files in the buffer**

```
vim file1 file2
```

Launching this command, you can see only one file at the same time. To jump between the files you can use the following vim commands:

n(ext)	jumps to the next file
prev	jumps to the previous file

- **Opening many files in several tabs**

```
vim -p file1 file2 file3
```

All files will be opened in tabs instead of hidden buffers. The tab bar is displayed on the top of the editor. You can also open a new tab with file *filename* when you're already in Vim in the normal mode with command:

```
:tabe filename
```

To manage tabs you can use the following vim commands:

:tabn[ext] (command-line command)	Jumps to the next tab
gt (normal mode command)	
:tabp[revious] (command-line command)	Jumps to the previous tab
gT (normal mode command)	
ngT (normal mode command)	Jumps to a specific tab index n= index of tab (starting by 1)
:tabc[lose] (command-line command)	Closes the current tab

- **Open multiple files splitting the window**
splits the window horizontally

```
vim -o file1 file2
```

You can also split the window horizontally, opening the file *filename*, when you're already in Vim in the normal mode with command:

```
:sp[lit] filename
```

splits the window vertically

```
vim -O file1 file2
```

You can also split the window vertically, opening the file *filename*, when you're already in Vim in the normal mode with command:

```
:vs[plit] filename
```

Management of the windows can be done, staying in the normal mode of Vim, using the following commands:

Ctrl+w <cursor-keys>	Jumps between windows
Ctrl+w [hjkl]	
Ctrl+w Ctrl+[hjkl]	
Ctrl+w w	Jumps to the next window
Ctrl+w Ctrl+w	
Ctrl+w W	Jumps to the previous window
Ctrl+w p	Jumps to the last accessed window
Ctrl+w Ctrl+p	
Ctrl+w c	Closes the current window
:clo[se]	
Ctrl+w o	Makes the current window the only one and closes all other ones
:on[ly]	

Appendix C

Gnu Project Debugger (GDB)

To use gdb you need to do the following 2 steps:

1. Compile the program with **-g** option, as follow:

```
gcc -g -o test test.c
```

2. Call gdb on the program you want to debug, as follow:

```
gdb test
```

3. Call *run* inside gdb, to run the program. You can add also command line arguments just writing them after run in the same line.
4. Call *quit* inside GDB to terminate the session

C.1 GDB commands

C.1.1 Breakpoints

break name_function	Set breakpoint on function called *name_function*
break example.c:name_function	Set breakpoint on function called *name_function* in file example.c
break XX	Set breakpoint at line numbered XX
break or **b**	Set breakpoint at line in which the program has already failed
break example.c:XX	Set breakpoint at line numbered XX in file example.c
clear XX	Remove breakpoint at line numbered XX
watch name_variable	Program will stop whenever the variable *name_variable* changes
step	Step into a function call
next or **n**	Step over a function call
bt	Print backtrace of the entire stack
up [count]	Select the previous (outer) stack frame or one of the frames preceding it (count frames up).
ENTER	Repeat the last command
continue or **c**	Continue until the next breakpoint or watchpoint is reached

C.1.2 Conditional breakpoints

Breakpoint with a condition statement. This is usefull, because you could insert condition also directly in the code but doing this you could add bugs that weren't before. A conditional breakpoint is made by adding if condition after the break statement in GDB, as follows:

```
break example.c:60 if (x > 255)
```

There are also conditional watchpoints made by typing sentences like the folllowing:

```
watch x > 10
```

In this case the watchpoint will be set on x and the program stops when x reaches reaches the value *0x11*. It can be usefull on multithreading.

C.1.3 Examine memory

To examine the memory you need to call the command *x* in one of the following ways:

```
x/nfu addr
x addr
x
```

where n, f, and u are all optional parameters that specify how much memory to display and how to format it; addr is an expression giving the address where you want to start displaying memory. If you use defaults for nfu, you need not type the slash '/'. Several commands set convenient defaults for addr. There are other commands

n	The repeat count is a decimal integer; the default is 1. It specifies how much memory (counting by units u) to display. If a negative number is specified, memory is examined backward from addr.
f	The display format is one of the formats used by print ('x', 'd', 'u', 'o', 't', 'a', 'c', 'f', 's') and in addition 'i' (for machine instructions). The default is 'x' (hexadecimal) initially. The default changes each time you use either x or print
u	Unit size (default = *w* except for *s* format that is *b*) **b** = bytes **w** = words (4B) **h** = halfwords (2B) **g** = giant words (8B)

used to examine and to set variable values to something. These are:

print name_variable **p name_variable**	Print the value of variable called *name_variable*
list	Show all the code in the file
set var name_variable=value	Set the value of the variable *name_variable* equal to *value*

C.1.4 Automate tasks in gdb

You can insert all the commands that you want to launch on gdb in a file *init.gdb* and then pass it to the program thanks to the option *-x*, as follows:

```
gdb test -x init.gdb
```

C.1.5 Debugging with fork() and exec()

set follow-fork-mode child	Specify that GDB needs to follow the child process after the fork() call in the program.
set follow-exec-mode new	Specify that GDB needs to follow the new program called by exec.

C.1.6 Debugging with multiple threads

info threads	Show the current threads in the program.
thread num	Switch to the execution made by thread with number *num*.
thread apply all *command*	Command is applied on all the threads.

Appendix D

Code

D.1 Endianness

```
int is_little_endian()
{
    int i=1;
  char* p = (char*) &i;

  return ((int) *p) == i;
}

short int htons(short int num)
{
    int size = sizeof(num);
    short int num2 = 0;
    int i;
    unsigned char* p1 = (unsigned char*) &num;
    unsigned char* p2 = (unsigned char*) &num2;

    if(is_little_endian)
    {
        for(i=0; i<size; i++)
            p2[i]=p1[size-i-1];

        return num2;
    }
    else
        return num;
}

int htonl(int num)
{
    int size = sizeof(num);
    int num2 = 0;
    int i;
    unsigned char* p1 = (unsigned char*) &num;
    unsigned char* p2 = (unsigned char*) &num2;

    if(is_little_endian)
    {
        for(i=0; i<size; i++)
            p2[i]=p1[size-i-1];

        return num2;
    }
    else
        return num;
}
```

D.2 HTTP

D.2.1 HEX to DEC convertion

```
char hex2dec(char c)
{
    printf("%c",c);
    switch(c)
    {
          case '0' ... '9':
              //printf("%c", c);
              c = c - '0';
              break;

          case 'A' ... 'F':
              //printf("%c", c);
              c = c - 'A' +10;
              break;

          case 'a' ... 'f':
              //printf("%c", c);
              c = c - 'a' +10;
              break;

          default:
              return -1;
    }

    return c;
}
```

D.2.2 Web Client

D.2.2.1 HTTP/0.9

```c
#include "net_utility.h"
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

struct sockaddr_in server;

int main(int argc, char ** argv)
{
    int sd;
    int t;
    int size;
    char request[100];
    char response[1000000];

    unsigned char ipaddr[4] = {216,58,211,163};

    if(argc>3)
        control(-1, "Too many arguments");

    //Initialization of TCP socket for IPv4 protocol
    sd = socket(AF_INET, SOCK_STREAM, 0);
    control(sd, "Socket failed \n");

    //Definition of IP address + Port of the server
    server.sin_family=AF_INET;
    server.sin_port = htons(80);

    if(argc>1)
    {
        server.sin_addr.s_addr=inet_addr(argv[1]);
        //or inet_aton(argv[1], &server.sin_addr);

        if(argc==3)
            server.sin_port = htons(atoi(argv[2]));
    }
    else
    {
        server.sin_addr.s_addr = *(uint32_t *) ipaddr;
        server.sin_port = htons(80);
    }

    //Connect to remote server
    t = connect(sd, (struct sockaddr *)&server, sizeof(server));
    control(t, "Connection failed \n");

    //Writing on socket (Sending request to server)
    sprintf(request, "GET /\r\n");
    size = my_strlen(request);
    t = write(sd, request, size);
    control(t, "Write failed\n");

    //Reading the response
    for(size=0; (t=read(sd, response+size, 1000000-size))>0; size=size+t);
    control(t, "Read failed \n");
    print_body(response, size, 0);

    return 0;
}
```

D.2.2.2 HTTP/1.0

```
#include "wc10.h"
#include "net_utility.h"

#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>

struct sockaddr_in server;
header h[30];

int main(int argc, char ** argv)
{
    int sd;
    int t;
    int i;
    int j;
    int k;
    int status_length;
    int size;
    int code;
    int body_length;
    char request[100];
    char response[1000000];
    char *website;
    char *status_tokens[3];
    unsigned char ipaddr[4] = {216, 58, 208, 131};

    if(argc>3)
    {
        control(-1,"Too many arguments");
    }

    //Initialization of TCP socket for IPv4 protocol
    sd = socket(AF_INET, SOCK_STREAM, 0);
    control(sd, "Socket failed \n");

    //Definition of IP address + Port of the server
    server.sin_family=AF_INET;
    server.sin_port = htons(80);

    if(argc>1)
    {
        server.sin_addr.s_addr=inet_addr(argv[1]);

        if(argc==3)
            server.sin_port = htons(atoi(argv[2]));
    }
    else
    {
        server.sin_addr.s_addr = *(uint32_t *) ipaddr;
        server.sin_port = htons(80);
    }

    //Connect to remote server
    t = connect(sd, (struct sockaddr *)&server, sizeof(server));
    control(t, "Connection failed \n");

    //Writing on socket (Sending request to server)
    sprintf(request, "GET http://www.google.com/ HTTP/1.0\r\nHost:www.google.com\r\
    nConnection:close\r\n\r\n");
    size = my_strlen(request);
    t = write(sd, request, size);
```

```
68      control(t, "Write failed\n");
69
70      j = 0;
71      k = 0;
72      h[k].name= response;
73
74      //Parser of response (HEADER*STATUS LINE)
75      while(read(sd, response+j, 1))
76      {
77          if((response[j]=='\n') && (response[j-1]=='\r'))
78          {
79              response[j-1]=0;
80
81              if(h[k].name[0]==0)
82                  break;
83
84              h[++k].name = response+j+1;
85          }
86
87          if(response[j]==':' && h[k].value==0)
88          {
89              response[j]=0;
90              h[k].value=response+j+1;
91          }
92          j++;
93      }
94
95      //Status line parser
96      status_length = my_strlen(h[0].name);
97
98      status_tokens[0]=h[0].name;
99      i=1;
100     k=1;
101     for(i=0; i<status_length && k<3; i++)
102     {
103         if(h[0].name[i]==' ')
104         {
105             h[0].name[i]=0;
106             status_tokens[k]=h[0].name+i+1;
107             k++;
108         }
109     }
110
111
112     printf(LINE);
113     printf("Status line:\n");
114     printf(LINE);
115
116     printf("HTTP version: %30s\n", status_tokens[0]);
117     code = atoi(status_tokens[1]);
118     printf("HTTP code:    %30d\n", code);
119     printf("HTTP version: %30s\n", status_tokens[2]);
120     printf(LINE);
121
122     //Analysis of header values
123     website=NULL;
124     for(i=1; h[i].name[0]; i++)
125     {
126         if(!strcmp(h[i].name, "Content-Length"))
127             body_length = atoi(h[i].value);
128
129         if(!strcmp(h[i].name, "Location") && code>300 && code<303)
130             website=h[i].value;
131
132         printf("Name=%s ----> Value=%s\n",h[i].name, h[i].value);
133     }
134
135     //Reading the response
136     if(body_length)
137         for(size=0; (t=read(sd, response+j+size, body_length-size))>0; size+=t);
```

```
138     else
139         for(size=0; (t=read(sd, response+j+size, 1000000-size))>0; size+=t);
140
141     control(t, "Read failed");
142
143     //Print the redirection
144     if(website!=NULL)
145     {
146         printf(LINE);
147         printf("\nRedirection:       %s \n\n", website);
148     }
149
150     //Print the response
151     print_body(response, size, j);
152
153     return 0;
154 }
```

D.2.2.3 HTTP/1.1

```c
#include "net_utility.h"
#include "wc11.h"

#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

struct sockaddr_in server;
header h[30];

int main(int argc, char ** argv)
{
    int sd;
    int t;
    int i;
    int k;
    int size;
    int header_size;
    int body_length=0;
    char request[100];
    char response[1000000];
    char entity[1000000];
    char *website=NULL;
    char *status_tokens[3];
    unsigned char ipaddr[4] = {192,168,1,81};

    if(argc>3)
    {
        perror("Too many arguments");
        return 1;
    }

    i=0;
    while(i<3)
    {
        //Initialization of TCP socket for IPv4 protocol
        sd = socket(AF_INET, SOCK_STREAM, 0);
        control(sd, "Socket failed\n");

        //Definition of IP address + Port of the server
        server.sin_family=AF_INET;
        server.sin_port = htons(80);

        if(argc>1)
        {
            server.sin_addr.s_addr=inet_addr(argv[1]);
            //or inet_aton(argv[1], &server.sin_addr);

            if(argc==3)
                server.sin_port = htons(atoi(argv[2]));
        }
        else
        {
            server.sin_port = htons(80);
            server.sin_addr.s_addr = *(uint32_t *) ipaddr;
        }

        //Connect to remote server
        t = connect(sd, (struct sockaddr *)&server, sizeof(server));
        control(t, "Connection failed \n");

```

```
69          //Writing on socket (Sending request to server)
70          sprintf(request, "GET␣/␣HTTP/1.1\r\nHost:www.google.com\r\n\r\n");
71          size = my_strlen(request);
72          t = write(sd, request, size);
73          control(t, "Write␣failed␣\n");
74
75          //Parsing the response (HEADER + STATUS LINE)
76          parse_header(sd, response, status_tokens, &header_size);
77
78          //Parsing header values
79          analysis_headers(status_tokens, h, &body_length, website);
80
81          //Read body of the response
82          body_acquire(sd, body_length, entity, &size);
83          print_body(entity, size, 0);
84          i++;
85
86           for(k=1; k<30 && h[k].name[0]; k++)
87              h[k].value=0;
88      }
89
90      return 0;
91  }
92
93  void parse_header(int sd, char* response, char** status_tokens, int* header_size)
94  {
95      //Parsing response (HEADER+STATUS LINE)
96      int j = 0;
97      int k = 0;
98      h[k].name= response;
99
100     while(read(sd, response+j, 1))
101     {
102         if((response[j]=='\n') && (response[j-1]=='\r'))
103         {
104             response[j-1]=0;
105
106             if(h[k].name[0]==0)
107                 break;
108
109             h[++k].name = response+j+1;
110         }
111
112         if(response[j]==':' && h[k].value==0)
113         {
114             response[j]=0;
115             h[k].value=response+j+1;
116         }
117         j++;
118     }
119
120     //Parsing Status Line
121     *header_size = k;
122     status_tokens[0]=h[0].name;
123     j=1;
124     k=1;
125
126     for(j=0; k<3; j++)
127     {
128         if(h[0].name[j]=='␣')
129         {
130             h[0].name[j]=0;
131             status_tokens[k++]=h[0].name+j+1;
132         }
133     }
134 }
135
136 void analysis_headers(char **status_tokens, header* h, int* body_length, char* website)
137 {
138     int code;
```

```
139     int i;
140
141     printf("\n");
142     printf(LINE);
143     printf(LINE);
144     printf("                    HEADERS\n");
145     printf(LINE);
146     printf("Status line\n");
147     printf("HTTP version: %30s\n", status_tokens[0]);
148     code = atoi(status_tokens[1]);
149     printf("HTTP code:    %30d\n", code);
150     printf("HTTP comment: %30s\n", status_tokens[2]);
151     printf(LINE);
152
153     website=NULL;
154     for(i=1; h[i].name[0]; i++)
155     {
156         if(!strcmp(h[i].name, "Content-Length"))
157             (*body_length) = atoi(h[i].value);
158
159         if(!strcmp(h[i].name, "Location") && code>300 && code<303)
160             website=h[i].value;
161
162         if(!strcmp(h[i].name, "Transfer-Encoding") && !strcmp(h[i].value," chunked"))
163             (*body_length)=-1;
164
165         printf("Name= %s -----> Value= %s\n",h[i].name, h[i].value);
166     }
167     printf(LINE);
168     printf("\n\n");
169 }
170
171
172 void body_acquire(int sd, int body_length, char* entity, int *size)
173 {
174     char c;
175     int t;
176     int chunk_size;
177     int is_size;
178
179     printf(LINE);
180     printf(LINE);
181     if(body_length>0)
182     {
183         printf("Reading of HTTP/1.0 (Content-length specified)\n");
184         for((*size)=0; (t=read(sd, entity+(*size), body_length-(*size)))>0; (*size)+=t);
185     }
186     if(body_length<0)
187     {
188         printf("Reading of HTTP/1.1 (chunked read)\n");
189         printf(LINE);
190         body_length=0;
191
192         do
193         {
194             chunk_size=0;
195             printf("HEX chunck size: ");
196
197             is_size=1;
198             while((t=read(sd, &c, 1))>0)
199             {
200                 if(c=='\n')
201                     break;
202                 else if(c=='\r')
203                     continue;
204                 else if(is_size)
205                 {
206                     c = hex2dec(c);
207
208                     if(c==-1)
```

```
209                         is_size=0;
210                     else
211                         chunk_size = chunk_size*16+c;
212                 }
213             }
214
215             control(t, "Chunk body read failed");
216
217             printf("\nChunk size: %d\n",chunk_size);
218             for((*size)=0; (t=read(sd, entity+body_length+(*size), chunk_size-(*size)))>0;
    (*size)+=t);
219
220             read(sd, &c, 1);
221             read(sd, &c, 1);
222
223             body_length+=chunk_size;
224             printf(LINE);
225         }
226         while(chunk_size>0);
227
228         (*size)=body_length;
229         printf("Size: %10d\n", *size);
230     }
231     else if(body_length==0)
232     {
233         printf("Reading of HTTP/0.9 (no Content-length specified)\n");
234         for(*size=0; (t=read(sd, entity+(*size), 1000000-(*size)))>0; (*size)+=t);
235     }
236     printf(LINE);
237     printf("\n\n");
238 }
```

D.2.2.4 Caching using HEAD and Last-Modified

```c
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h> /* superset of previous */
#include <arpa/inet.h>
#include <stdio.h>
#include <stdint.h>
#include <errno.h>
#include <assert.h>

#define __USE_XOPEN
#include <time.h>
//#define USE_GMT

struct sockaddr_in server;

struct headers {
    char *n;
    char *v;
}h[30];

#define LINE_SIZE 100

int main(int argc, char** argv)
{
    int s,t,size,i,j,k;
    char request[100],response[1000000];
    unsigned char ipaddr[4]={93,184,216,34};
    int bodylength=0;
    char resource[50];
    char resource_path[50] = "./cache/";
    FILE* f;
    int head=0;
    char line[LINE_SIZE];
    int is_updated=0;
    time_t download_time;
    time_t last_time;
    char *version, *code, *comment;

    s = socket(AF_INET, SOCK_STREAM, 0);
    if ( s == -1) { printf("Errno = %d\n", errno); perror("Socket Failed"); return 1; }

    server.sin_family = AF_INET;
    server.sin_port = htons(80);
    server.sin_addr.s_addr = *(uint32_t *)ipaddr;

    t = connect(s, (struct sockaddr *)&server, sizeof(server));
    if ( t == -1) { perror("Connect Failed"); return 1; }

    strcpy(resource, argv[1]);
    for(i=0; i<strlen(argv[1]); i++)
    {
        if(argv[1][i]=='/')
            resource[i]='_';
    }

    strcat(resource_path, resource);
    if((f=fopen(resource_path, "r"))!=NULL)
    {
        sprintf(request,"HEAD %s HTTP/1.0\r\nHost:www.example.com\r\n\r\n", argv[1]);
        head=1;
    }
    else
        sprintf(request,"GET %s HTTP/1.0\r\nHost:www.example.com\r\n\r\n", argv[1]);

    for(size=0;request[size];size++);
```

```
69      t=write(s,request,size);
70      if ( t == -1 ) { perror("Write failed"); return 1; }
71
72      j=0;k=0;
73    h[k].n = response;
74    while(read(s,response+j,1))
75      {
76           if((response[j]=='\n') && (response[j-1]=='\r'))
77           {
78                response[j-1]=0;
79                if(h[k].n[0]==0) break;
80                h[++k].n=response+j+1;
81           }
82
83           if(response[j]==':' && (h[k].v==0) )
84           {
85                response[j]=0;
86                h[k].v=response+j+1;
87           }
88
89           j++;
90    }
91
92      char *last_modified = NULL;
93
94      version = h[0].n;
95      for(i=0; h[0].n[i]!=' '; i++);
96      h[0].n[i]=0;
97
98      i++;
99      code = h[0].n+i;
100     for(; h[0].n[i]!=' '; i++);
101     h[0].n[i]=0;
102
103     comment = h[0].n+i+1;
104
105     printf("%s %s %s\n", version, code, comment);
106     for(i=1;h[i].n[0];i++)
107     {
108     if (!strcmp(h[i].n,"Content-Length"))
109               bodylength = atoi(h[i].v);
110          else if (!strcmp(h[i].n, "Last-Modified"))
111               last_modified = h[i].v;
112
113     printf("%s:%s\n",h[i].n,h[i].v);
114     }
115
116     if(head && last_modified!=NULL)
117     {
118          //fopen works well
119          struct tm tm, tm2;
120          memset(&tm, 0, sizeof(tm));
121          strptime(last_modified, "%a, %d %b %Y %H:%M:%S %Z", &tm);
122
123          #ifdef USE_GMT
124               last_time = timegm(&tm);
125          #else
126               last_time = mktime(&tm);
127          #endif
128
129          time_t cache_time;
130          char date[100];
131          fgets(date, 100, f);
132          strptime(date, "%a, %d %b %Y %H:%M:%S %Z", &tm2);
133
134          #ifdef USE_GMT
135               cache_time = timegm(&tm2);
136          #else
137               cache_time = mktime(&tm2);
138          #endif
```

```
139
140         if(cache_time<last_time)
141         {
142             shutdown(s, SHUT_RDWR);
143             close(s);
144             s = socket(AF_INET, SOCK_STREAM, 0);
145
146             if ( s == -1)
147             {
148                 printf("Errno = %d\n", errno);
149                 perror("Socket Failed");
150                 return 1;
151             }
152
153             server.sin_family = AF_INET;
154             server.sin_port = htons(8083);
155             server.sin_addr.s_addr = *(uint32_t *)ipaddr;
156             // WRONG : server.sin_addr.s_addr = (uint32_t )*ipaddr
157
158             t = connect(s, (struct sockaddr *)&server, sizeof(server));
159             if ( t == -1) { perror("Connect Failed"); return 1; }
160
161             sprintf(request,"GET %s HTTP/1.0\r\nHost:192.168.1.81\r\n\r\n", argv[1]);
162             write(s, request, strlen(request));
163
164             for(i=0; h[i].n[0]; h[i++].v=0);
165
166             j=0;k=0;
167             h[k].n = response;
168             while(read(s,response+j,1))
169             {
170                 printf("%c", response[j]);
171                 if((response[j]=='\n') && (response[j-1]=='\r'))
172                 {
173                     response[j-1]=0;
174                     if(h[k].n[0]==0) break;
175                     h[++k].n=response+j+1;
176                 }
177
178                 if(response[j]==':' && (h[k].v==0) )
179                 {
180                     response[j]=0;
181                     h[k].v=response+j+1;
182                 }
183
184                 j++;
185             }
186         }
187         else
188             is_updated=1;
189
190         fclose(f);
191     }
192
193     if(!is_updated)
194     {
195         //fopen works bad
196         assert((f= fopen(resource_path, "w"))!=NULL);
197
198         if (bodylength) // we have content-length
199         for(size=0; (t=read(s,response+size,bodylength-size))>0;size=size+t);
200       else
201         for(size=0; (t=read(s,response+size,1000000-size))>0;size=size+t);
202
203       if ( t == -1 ) { perror("Read failed"); return 1; }
204
205         response[size]=0;
206         char down_time[40];
207         time_t download_time = time(0);
208         struct tm* tm;
```

```
209 ||
210 ||          #ifdef USE_GMT
211 ||              tm=gmtime(&download_time);
212 ||          #else
213 ||              tm=localtime(&download_time);
214 ||          #endif
215 ||
216 ||          strftime(down_time, 40, "%a, %d %b %Y %H:%M:%S %Z", tm);
217 ||          fprintf(f, "%s\n%s", down_time, response);
218 ||          fclose(f);
219 ||      }
220 ||      else
221 ||      for(size=0; (t=read(s,response+size,1000000-size))>0;size=size+t);
222 ||
223 ||      return 0;
224 || }
```

D.2.2.5 Caching using If-Modified-Since

```c
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h> /* superset of previous */
#include <arpa/inet.h>
#include <stdio.h>
#include <stdint.h>
#include <errno.h>
#include <assert.h>

#define __USE_XOPEN
#include <time.h>
//#define USE_GMT

struct sockaddr_in server;

struct headers {
    char *n;
    char *v;
}h[30];

#define LINE_SIZE 100

int main(int argc, char** argv)
{
    int s,t,size,i,j,k;
    char *version, *code, *comment;
    char request[100],response[1000000];
    unsigned char ipaddr[4]={93,184,216,34};
    int bodylength=0;
    char resource[50];
    char resource_path[50] = "./cache/";
    FILE* f;
    char line[LINE_SIZE];
    int is_updated=0;
    time_t download_time;
    time_t last_time;

    s = socket(AF_INET, SOCK_STREAM, 0);
    if ( s == -1) { printf("Errno = %d\n", errno); perror("Socket Failed"); return 1; }

    server.sin_family = AF_INET;
    server.sin_port = htons(8083);
    server.sin_addr.s_addr = *(uint32_t *)ipaddr;
    // WRONG : server.sin_addr.s_addr = (uint32_t )*ipaddr

    t = connect(s, (struct sockaddr *)&server, sizeof(server));
    if ( t == -1) { perror("Connect Failed"); return 1; }

    strcpy(resource, argv[1]);
    for(i=0; i<strlen(argv[1]); i++)
    {
        if(argv[1][i]=='/')
            resource[i]='_';
    }

    strcat(resource_path, resource);
    if((f=fopen(resource_path, "r"))!=NULL)
    {
        char date[100];
        fgets(date, 100, f);
        fclose(f);

        sprintf(request,"GET %s HTTP/1.0\r\nHost:www.example.com\r\nIf-Modified-Since:%s\r\n\r\n", argv[1], date);
    }
```

```
68      else
69      {
70           sprintf(request,"GET␣%s␣HTTP/1.0\r\nHost:www.example.com\r\n\r\n", argv[1]);
71      }
72
73      for(size=0;request[size];size++);
74      t=write(s,request,size);
75      if ( t == -1 ) { perror("Write␣failed"); return 1; }
76
77      j=0;k=0;
78    h[k].n = response;
79    while(read(s,response+j,1))
80      {
81           if((response[j]=='\n') && (response[j-1]=='\r'))
82           {
83                response[j-1]=0;
84                if(h[k].n[0]==0) break;
85                h[++k].n=response+j+1;
86           }
87
88           if(response[j]==':' && (h[k].v==0) )
89           {
90                response[j]=0;
91                h[k].v=response+j+1;
92           }
93
94           j++;
95    }
96
97      char *last_modified = NULL;
98      version = h[0].n;
99      for(i=0; response[i]!='␣'; i++);
100     response[i]=0;
101
102     code = h[0].n+i+1;
103     for(i=i+1; response[i]!='␣'; i++);
104     response[i]=0;
105
106     comment = h[0].n+i+1;
107     printf("%s␣%s␣%s\n", version, code, comment);
108
109   for(i=1;h[i].n[0];i++)
110     {
111     if (!strcmp(h[i].n,"Content-Length"))
112              bodylength = atoi(h[i].v);
113
114     printf("%s:%s\n",h[i].n,h[i].v);
115     }
116
117     if(strcmp(code, "304"))
118     {
119          //fopen works bad
120          assert((f= fopen(resource_path, "w"))!=NULL);
121
122          if (bodylength) // we have content-length
123          for(size=0; (t=read(s,response+size,bodylength-size))>0;size=size+t);
124        else
125          for(size=0; (t=read(s,response+size,1000000-size))>0;size=size+t);
126
127       if ( t == -1 ) { perror("Read␣failed"); return 1; }
128
129          response[size]=0;
130
131
132          char down_time[40];
133          time_t download_time = time(0);
134          struct tm* tm;
135
136          #ifdef USE_GMT
137               tm=gmtime(&download_time);
```

```
138             #else
139                 tm=localtime(&download_time);
140             #endif
141 
142             strftime(down_time, 40, "%a, %d %b %Y %H:%M:%S %Z", tm);
143             fprintf(f, "%s\n%s", down_time, response);
144 
145             fclose(f);
146         }
147 
148         return 0;
149 }
```

D.2.3 Web Proxy

D.2.3.1 Standard version with HTTP and HTTPS

```
#include "wp.h"
#include "net_utility.h"

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <signal.h>
#include <netdb.h>

struct sockaddr_in local, remote;
struct hostent* he;

int main(int argc, char** argv)
{
    char request[2000];
    char *method, *path, *version;
    int sd, sd2;
    int t;
    socklen_t len;
    int yes = 1;

    //Initialization of TCP socket for IPv4 protocol between client and proxy
    sd = socket(AF_INET, SOCK_STREAM, 0);
    control(sd, "Socket failed \n");

    //Bind the server to a specific port
    local.sin_family=AF_INET;
    local.sin_port = htons(atoi(argv[1])); //we need to use a port not in use
    local.sin_addr.s_addr = 0; //By default

    //Reuse the same IP already bind to other program
    setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));
    t = bind(sd, (struct sockaddr*) &local, sizeof(struct sockaddr_in));
    control(t, "Bind failed \n");

    //Queue of pending clients that want to connect
    t = listen(sd, QUEUE_MAX);
    control(t, "Listen failed \n");

    if(t==-1)
    {
        printf("Errno: %d\n", errno);
        perror("Listen Failed");
        return 1;
    }

    while(1)
    {
        remote.sin_family = AF_INET;
        len = sizeof(struct sockaddr_in);

        //Accept the new request and create its socket
        sd2 = accept(sd, (struct sockaddr*) &remote, &len);
        control(sd2, "Accept failed \n");

        //A child manages the single request
        if(!fork())
        {
            //Read the request of the client
```

```
67              t = read(sd2, request, 1999);
68              request[t]=0;
69
70              //Parser of request line
71              request_line(request, &method, &path, &version);
72
73              //Manage the response to the request
74              manage_request(method, path, version, sd2);
75
76              //Shutdown the socket created with the specific client
77              shutdown(sd2, SHUT_RDWR);
78              close(sd2);
79              exit(0);
80          }
81      }
82  }
83
84
85  void request_line(char* request, char** method, char** path, char** version)
86  {
87      int i;
88      *method = request;
89
90      for(i=0; request[i]!='␣'; i++);
91
92      request[i]=0;
93      *path=request+i+1;
94
95      for(; request[i]!='␣'; i++);
96
97      request[i]=0;
98      *version=request+i+1;
99
100     for(; (request[i]!='\n' || request[i-1]!='\r') ; i++);
101
102     request[i-1]=0;
103 }
104
105 void manage_request(char* method, char* path, char* version, int sd2)
106 {
107     char request2[2000], response[2000], response2[2000];
108     int t;
109
110     printf("Method:␣␣%s\n", method);
111     printf("Path:␣␣%s\n", path);
112     printf("Version:␣␣%s\n", version);
113
114     if(!strncmp(method,"GET", 3)) //GET request
115     {
116         printf("\n\nGET\n\n");
117         char *scheme, *host, *resource;
118         parser_path(path, &scheme, &host, &resource);
119
120         int sd3 = connect2server(host, "80"); //HTTP service
121
122         //Write the request to the server
123         sprintf(request2, "GET␣/%s␣HTTP/1.1\r\nHost:%s\r\nConnection:close\r\n\r\n",
    resource, host);
124         write(sd3, request2, strlen(request2));
125         printf("request2:␣%s\n\n", request2);
126
127         //Forward response from server to client
128         while((t=read(sd3, response2, 2000)))
129         {
130             write(sd2, response2,t);
131         }
132
133         //Shutdown the socket created with the server
134         shutdown(sd3, SHUT_RDWR);
135         close(sd3);
```

```
136     }
137     else if(!strncmp(method,"CONNECT", 7))
138     {
139         printf("\n\nCONNECT\n\n");
140         char *host, *port;
141         parser_connect(path, &host, &port);
142 
143         int sd3 = connect2server(host, port);
144 
145         sprintf(response, "HTTP/1.1 200 Established\r\n\r\n");
146 
147         write(sd2, response, strlen(response));
148 
149         int pid;
150 
151         if((pid=fork())==0)//child to forward data from client to server
152         {
153             //Forwaring request from client to server
154             while((t=read(sd2, request2, 2000)))
155             {
156                 printf("C2P>> t: %d\n", t);
157                 write(sd3, request2, t);
158             }
159 
160             exit(0);
161         }
162         else if(pid>0)//parent to forward data from server to client
163         {
164             //Forwarding response from server to client
165             while((t=read(sd3, response2, 2000)))
166             {
167                 printf("S2P>> t: %d\n", t);
168                 write(sd2, response2, t);
169             }
170 
171             //Kill child (process that manages data from client to server)
172             kill(pid,SIGTERM);
173 
174             //Shutdown the socket created with the server
175             shutdown(sd3, SHUT_RDWR);
176             close(sd3);
177         }
178         else
179             printf("\n\nERROR: creation of process\n\n");
180     }
181 }
182 
183 void parser_path(char* path, char** scheme, char** host, char** resource)
184 {
185     //http://www.ciao.it/path
186     *scheme = path;
187 
188     int i=0;
189     for(; path[i]!=':'; i++);
190     path[i]=0;
191 
192     *host = path+i+3;
193     for(i=i+3; path[i]!='/'; i++);
194     path[i]=0;
195 
196     *resource = path +i+1;
197 
198     printf("Scheme=%s  Host=%s  Resource=%s\n", *scheme, *host, *resource);
199 }
200 
201 void parser_connect(char* path, char** host, char** port)
202 {
203     int i=0;
204 
205     //www.ciao.it:8080
```

```
206     printf("\n\narrivato\n\n");
207     *host = path;
208
209     for(; path[i]!=':'; i++);
210     path[i]=0;
211
212     *port = path+i+1;
213
214     printf("Host=%s  Port=%s\n", *host, *port);
215 }
216
217 int connect2server(char* host, char* port)
218 {
219     struct sockaddr_in server;
220     int t, sd3;
221
222     //Resolve name to IP address using DNS
223     struct hostent* he;
224     he = gethostbyname(host);
225
226     if(he == NULL)
227     {
228         perror("Gethostbyname Failed");
229         exit(1);
230     }
231
232     //Initialization of TCP socket for IPv4 protocol between proxy and server
233     sd3 = socket(AF_INET, SOCK_STREAM, 0);
234     control(sd3, "Socket failed 2\n");
235
236     //Connect proxy to remote server
237     server.sin_family = AF_INET;
238     server.sin_port = htons((unsigned short) atoi(port));
239     server.sin_addr.s_addr = * (uint32_t*) he->h_addr;
240
241     t = connect(sd3, (struct sockaddr*) &server, sizeof(struct sockaddr_in));
242     control(t, "Connection failed 2\n");
243
244     return sd3;
245 }
```

D.2.3.2 Double type of connections

```
#include <sys/types.h>
#include <signal.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <netdb.h>

#include "net_utility.h"

#define BLUE "\033[1;34m"
#define CYAN "\033[1;36m"
#define DEFAULT "\033[0m"
#define GREEN "\033[1;32m"
#define MAGENTA "\033[1;35m"
#define RED "\033[1;31m"
#define YELLOW "\033[1;33m"

struct hostent * he;
struct sockaddr_in local, remote, server;

char request[100000],response[100000],request2[100000],response2[100000];
char *method, *path, *version, *host, *scheme, *resource, *port;

struct headers {
char *n;
char *v;
}h[30];

int main()
{
    FILE *f;
    char command[100];
    int i,s,t,s2,s3,n,len,c,yes=1,j,k,pid;
    int chunked = 0;
    int keep_alive;
    char conn2server_type[30];
    int body_length=0;
    int size=0;
    int chunk_size;

    s = socket(AF_INET, SOCK_STREAM, 0);
    if ( s == -1) { perror("Socket Failed\n"); return 1;}

    local.sin_family=AF_INET;
    local.sin_port = htons(8080);
    local.sin_addr.s_addr = 0;

    setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int));
    t = bind(s,(struct sockaddr *) &local, sizeof(struct sockaddr_in));
    if ( t == -1) { perror("Bind Failed \n"); return 1;}

    t = listen(s,10);
    if ( t == -1) { perror("Listen Failed \n"); return 1;}

    while( 1 )
    {
        f = NULL;
        remote.sin_family=AF_INET;
        len = sizeof(struct sockaddr_in);

        s2 = accept(s,(struct sockaddr *) &remote, &len);
        if(fork()) continue;
        if (s2 == -1) {perror("Accept Failed\n"); return 1;}
```

```
69
70          j=0;k=0;
71          h[k].n = request;
72          while(read(s2,request+j,1))
73          {
74               if((request[j]=='\n') && (request[j-1]=='\r'))
75               {
76                    request[j-1]=0;
77
78                    if(h[k].n[0]==0)
79                         break;
80
81                    h[++k].n=request+j+1;
82               }
83               if(request[j]==':' && (h[k].v==0) && k!=0)
84               {
85                    request[j]=0;
86                    h[k].v=request+j+1;
87               }
88               j++;
89          }
90
91          printf("%s\n",request);
92          method = request;
93          for(i=0;(i<2000) && (request[i]!=' ');i++); request[i]=0;
94          path = request+i+1;
95          for(    ;(i<2000) && (request[i]!=' ');i++); request[i]=0;
96          version = request+i+1;
97          printf("Method = %s, path = %s, version = %s\n",method,path,version);
98
99          if(!strcmp(version, "HTTP/1.0"))
100         {
101              for(i=1; h[i].n[0]; i++)
102              {
103                   if(!strcmp(h[i].n, "Connection"))
104                   {
105                        if(!strcmp(h[i].v, "keep-alive"))
106                             keep_alive = 1;
107                        else if(!strcmp(h[i].v, "close"))
108                             keep_alive = 0;
109                        else
110                             keep_alive = -1;
111                   }
112              }
113         }
114         else if(!strcmp(version, "HTTP/1.1"))
115              keep_alive = 1;
116
117         //for(i=1; h[i].n[0]; i++)
118         //     printf("%s----->%s\n", h[i].n, h[i].v);
119
120         printf("Keep alive: %d\n", keep_alive);
121         if(keep_alive>0)
122              strcpy(conn2server_type,"close");
123         else if(keep_alive==0)
124              strcpy(conn2server_type,"keep-alive");
125
126         if(!strcmp("GET",method)) //it is a GET
127         {
128              //http://www.google.com/path
129              scheme=path;
130              for(i=0;path[i]!=':';i++); path[i]=0;
131              host=path+i+3;
132              for(i=i+3;path[i]!='/';i++); path[i]=0;
133              resource=path+i+1;
134              printf("Scheme=%s%s%s, host=%s%s%s, resource=%s%s%s\n", CYAN, scheme, DEFAULT,
     RED, host, DEFAULT, GREEN, resource, DEFAULT);
135
136              he = gethostbyname(host);
137              if (he == NULL) { printf("Gethostbyname Failed\n"); return 1;}
```

```
138             printf("Server address = %u.%u.%u.%u\n", (unsigned char ) he->h_addr[0],(
    unsigned char ) he->h_addr[1],(unsigned char ) he->h_addr[2],(unsigned char ) he->h_addr
    [3]);
139
140             s3=socket(AF_INET,SOCK_STREAM,0);
141             if(s3==-1){perror("Socket to server failed"); return 1;}
142
143             server.sin_family=AF_INET;
144             server.sin_port=htons(80);
145             server.sin_addr.s_addr=*(unsigned int*) (he->h_addr);
146             t=connect(s3,(struct sockaddr *)&server,sizeof(struct sockaddr_in));
147             if(t==-1){perror("Connect to server failed"); return 1;}
148
149             if(keep_alive>=0)
150             {
151                 sprintf(request2,"GET /%s HTTP/1.1\r\nHost:%s\r\nConnection:%s\r\n\r\n",
    resource, host, conn2server_type);
152                 write(s3,request2,strlen(request2));
153                 printf("%s%s%s\n", BLUE, request2, DEFAULT);
154                 //Read the answer of the server and forward it to client
155                 memset(h, 0, 30*sizeof(struct headers));
156
157                 j=0;k=0;
158                 h[k].n = response;
159
160                 while(read(s3,response+j,1))
161                 {
162                     if((response[j]=='\n') && (response[j-1]=='\r'))
163                     {
164                         response[j-1]=0;
165
166                         if(h[k].n[0]==0)
167                             break;
168
169                         h[++k].n=response+j+1;
170                     }
171
172                     if(response[j]==':' && (h[k].v==0) && k!=0)
173                     {
174                         response[j]=0;
175                         h[k].v=response+j+1;
176                     }
177                     j++;
178                 }
179
180                 sprintf(response2, "%s\r\n", h[0].n);
181                 printf("%s%s%s\n", CYAN, h[0].n, DEFAULT);
182                 write(s2, response2, strlen(response2));
183
184                 for(i=1; h[i].n[0]; i++)
185                 {
186                     if(!strcmp(h[i].n, "Content-Length"))
187                         body_length=atoi(h[i].v);
188                     else if(!strcmp(h[i].n, "Transfer-Encoding") && !strcmp(h[i].v, "
    chunked"))
189                         body_length = -1;
190                     else
191                     {
192                         sprintf(response2, "%s:%s\r\n", h[i].n, h[i].v);
193                         write(s2, response2, strlen(response2));
194                     }
195
196                     printf("%s%s:%s%s\n", YELLOW, h[i].n, DEFAULT, h[i].v);
197                 }
198
199                 if(keep_alive)//Keep-alive on client, close from server
200                 {
201                     sprintf(response, "Transfer-Encoding:chunked\r\n\r\n");
202                     write(s2, response, strlen(response));
203
```

```
204                     if(body_length<0)
205                     {
206                         char c;
207                         while((t=read(s3, &c, 1))!=0)
208                         {
209                             write(s2, &c, 1);
210                         }
211                     }
212                     else
213                     {
214                         if(body_length==0)
215                             body_length = 10000;
216
217                         for(size=0; (t=read(s3, response, body_length-size))>0; size+=t)
218                         {
219                             sprintf(response2, "%x\r\n", t);
220                             write(s2, response2, strlen(response2));
221                             write(s2, response, t);
222                             write(s2, "\r\n", 2);
223                         }
224                         write(s2, "0\r\n\r\n", 5);
225                     }
226                 }
227                 else
228                 {
229                     if(body_length>0)
230                     {
231                         sprintf(response2, "Content-Length:%d\r\n\r\n", body_length);
232                         write(s2, response2, strlen(response2));
233
234                         size=0;
235                         while((t=read(s3, response, body_length-size))>0)
236                         {
237                             write(s2, response, t);
238                             size+=t;
239                         }
240                     }
241                     else if(body_length<0)
242                     {
243                         printf("Chunked␣reading\n");
244                         int count=1;
245                         body_length=0;
246
247                         do
248                         {
249                             printf("Chunk␣%2d:", count);
250                             char c;
251                             chunk_size=0;
252                             int is_size=1;
253
254                             while((t=read(s3, &c, 1))>0)
255                             {
256                                 if(c=='\n')
257                                     break;
258                                 else if(c=='\r')
259                                     continue;
260                                 else if(is_size)
261                                 {
262                                     c=hex2dec(c);
263
264                                     if(c==-1)
265                                         is_size=0;
266                                     else
267                                         chunk_size = chunk_size*16 + c;
268                                 }
269                             }
270
271                             if(t==-1)
272                                 perror("line␣223");
273
```

```
274                             for(size=0; (t=read(s3, response+body_length+size, chunk_size-
    size))>0; size+=t);
275                             read(s3, &c, 1);
276                             read(s3, &c, 1);
277
278                             printf("\n");
279                             body_length+=chunk_size;
280                             count++;
281                         }
282                         while(chunk_size>0);
283
284                         sprintf(response2, "Content-Length:%d\r\n\r\n", body_length);
285                         write(s2, response2, strlen(response2));
286                         write(s2, response, body_length);
287                     }
288                 }
289
290             }
291             else
292             {
293                 sprintf(response2, "HTTP/1.1 400 Bad Request\r\n\r\n");
294                 write(s2, response2, strlen(response2));
295             }
296
297             shutdown(s3,SHUT_RDWR);
298             close(s3);
299         }
300         else if(!strcmp("CONNECT",method))  //CONNECT
301         {
302             host=path;
303             for(i=0;path[i]!=':';i++); path[i]=0;
304             port=path+i+1;
305             printf("host:%s, port:%s\n",host,port);
306
307             he = gethostbyname(host);
308             if (he == NULL) { printf("Gethostbyname Failed\n"); return 1;}
309             printf("Connecting to address = %u.%u.%u.%u\n", (unsigned char ) he->h_addr
    [0],(unsigned char ) he->h_addr[1],(unsigned char ) he->h_addr[2],(unsigned char ) he->
    h_addr[3]);
310             s3=socket(AF_INET,SOCK_STREAM,0);
311             if(s3==-1){perror("Socket to server failed"); return 1;}
312
313             server.sin_family=AF_INET;
314             server.sin_port=htons((unsigned short)atoi(port));
315             server.sin_addr.s_addr=*(unsigned int*) he->h_addr;
316             t=connect(s3,(struct sockaddr *)&server,sizeof(struct sockaddr_in));
317             if(t==-1){perror("Connect to server failed"); exit(0);}
318
319             sprintf(response,"HTTP/1.1 200 Established\r\n\r\n");
320             write(s2,response,strlen(response));
321
322             if(!(pid=fork()))
323             {
324                 //Child
325                 while(t=read(s2,request2,2000))
326                 {
327                     write(s3,request2,t);
328                     printf("CL >>>(%d)%s \n",t,host); //SOLO PER CHECK
329                 }
330
331                 exit(0);
332             }
333             else
334             {
335                 //Parent
336                 while(t=read(s3,response2,2000))
337                 {
338                     write(s2,response2,t);
339                     printf("CL <<<(%d)%s \n",t,host);
340                 }
```

```
341 ||
342 ||                  kill(pid,15);
343 ||                  shutdown(s3,SHUT_RDWR);
344 ||                  close(s3);
345 ||              }
346 ||          }
347 ||
348 ||          shutdown(s2,SHUT_RDWR);
349 ||          close(s2);
350 ||          exit(0);
351 ||      }
352 ||
353 ||      return 0;
354 || }
```

D.2.3.3 Blacklist

```
#include <sys/types.h>          /* See NOTES */
#include <signal.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netinet/ip.h> /* superset of previous */
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <netdb.h>

#include "net_utility.h"

#define SIZE_BLACK_LIST 3

struct hostent * he;
struct sockaddr_in local,remote,server;
char request[10000],response[2000],request2[2000],response2[2000];
char * method, *path, *version, *host, *scheme, *resource,*port;
char black_list[SIZE_BLACK_LIST][20] = {"www.google.com",
                                        "www.radioamatori.it",
                                        "www.youtube.com"};

struct headers {
    char *n;
    char *v;
}h[30];

int main()
{
    FILE *f;
    char *type, *sub_type;
    char command[100], c;
    int i,s,t,s2,s3,n,len,yes=1,j,k,pid,size, block=0;

    s = socket(AF_INET, SOCK_STREAM, 0);
    if ( s == -1)
    {
        perror("Socket Failed\n");
        return 1;
    }

    local.sin_family=AF_INET;
    local.sin_port = htons(8080);
    local.sin_addr.s_addr = 0;
    setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int));
    t = bind(s,(struct sockaddr *) &local, sizeof(struct sockaddr_in));
    if ( t == -1)
    {
        perror("Bind Failed \n");
        return 1;
    }

    t = listen(s,10);
    if ( t == -1)
    {
        perror("Listen Failed \n");
        return 1;
    }

    while( 1 )
    {
        f = NULL;
        remote.sin_family=AF_INET;
        len = sizeof(struct sockaddr_in);

        s2 = accept(s,(struct sockaddr *) &remote, &len);
```

```
69
70          if(fork()) continue; //<< MULTI PROCESS HADLING
71          if (s2 == -1)
72          {
73               perror("Accept Failed\n");
74               return 1;
75          }
76
77          // <---- ADDED HEADER PARSER
78          j=0;k=0;
79          h[k].n = request;
80          while(read(s2,request+j,1))
81          {
82               if((request[j]=='\n') && (request[j-1]=='\r'))
83               {
84                    request[j-1]=0;
85
86                    if(h[k].n[0]==0)
87                         break;
88
89                    h[++k].n=request+j+1;
90               }
91
92               if(request[j]==':' && (h[k].v==0) && k!=0)
93               {
94                    request[j]=0;
95                    h[k].v=request+j+1;
96               }
97
98               j++;
99          }
100
101     printf("%s",request);
102     method = request;
103     for(i=0;(i<2000) && (request[i]!=' ');i++); request[i]=0;
104     path = request+i+1;
105     for(    ;(i<2000) && (request[i]!=' ');i++); request[i]=0;
106     version = request+i+1;
107          printf("Method = %s, path = %s, version = %s\n",method,path,version);
108
109          if(!strcmp("GET",method))
110          {
111               // http://www.google.com/path
112       scheme=path;
113       for(i=0;path[i]!=':';i++); path[i]=0;
114       host=path+i+3;
115       for(i=i+3;path[i]!='/';i++); path[i]=0;
116       resource=path+i+1;
117       printf("Scheme=%s, host=%s, resource = %s\n", scheme,host,resource);
118
119               for(i=0; i<SIZE_BLACK_LIST; i++)
120               {
121                    if(!strcmp(host, black_list[i]))
122                    {
123                         block=1;
124                         break;
125                    }
126               }
127
128               he = gethostbyname(host);
129       if (he == NULL)
130               {
131                    printf("Gethostbyname Failed\n");
132                    return 1;
133               }
134
135       printf("Server address = %u.%u.%u.%u\n", (unsigned char ) he->h_addr[0],(unsigned
      char ) he->h_addr[1],(unsigned char ) he->h_addr[2],(unsigned char ) he->h_addr[3]);
136       s3=socket(AF_INET,SOCK_STREAM,0);
```

```
137         if(s3==-1)
138                 {
139                     perror("Socket to server failed");
140                     return 1;
141                 }
142 
143                 server.sin_family=AF_INET;
144         server.sin_port=htons(80);
145         server.sin_addr.s_addr=*(unsigned int*) he->h_addr;
146         t=connect(s3,(struct sockaddr *)&server,sizeof(struct sockaddr_in));
147                 if(t==-1)
148                 {
149                     perror("Connect to server failed");
150                     return 1;
151                 }
152 
153                 if(block)
154                 {
155                     sprintf(response2, "HTTP/1.1 401 Unauthorized\r\n\r\n");
156                     write(s2, response2, strlen(response2));
157                 }
158                 else
159                 {
160               sprintf(request2,"GET /%s HTTP/1.1\r\nHost:%s\r\nConnection:close\r\n\r\n",
        resource,host);
161                     write(s3,request2,strlen(request2));
162 
163                     while((t=read(s3, response2, 2000))>0)
164                         write(s2, response2, t);
165 
166                     if(t==-1)
167                     {
168                         perror("[PROXY ERROR] Reading server response");
169                         exit(1);
170                     }
171                 }
172 
173                 shutdown(s3,SHUT_RDWR);
174                 close(s3);
175         }
176         else if(!strcmp("CONNECT",method))
177             {
178                 // www.google.com:400
179         host=path;
180         for(i=0;path[i]!=':';i++); path[i]=0;
181         port=path+i+1;
182         printf("host:%s, port:%s\n",host,port);
183 
184                 for(i=0; i<SIZE_BLACK_LIST; i++)
185                 {
186                     if(!strcmp(host, black_list[i]))
187                     {
188                         block=1;
189                         break;
190                     }
191                 }
192 
193                 if(block)
194                 {
195                     sprintf(response2, "HTTP/1.1 401 Unauthorized\r\n\r\n");
196                     write(s2, response2, strlen(response2));
197                 }
198                 else
199                 {
200                     he = gethostbyname(host);
201                     if (he == NULL)
202                     {
203                         printf("Gethostbyname Failed\n");
204                         return 1;
205                     }
```

```
206
207                 printf("Connecting to address = %u.%u.%u.%u\n", (unsigned char ) he->h_addr
        [0],(unsigned char ) he->h_addr[1],(unsigned char ) he->h_addr[2],(unsigned char ) he->
        h_addr[3]);
208                 s3=socket(AF_INET,SOCK_STREAM,0);
209                 if(s3==-1)
210                 {
211                     perror("Socket to server failed");
212                     return 1;
213                 }
214
215                 server.sin_family=AF_INET;
216                 server.sin_port=htons((unsigned short)atoi(port));
217                 server.sin_addr.s_addr=*(unsigned int*) he->h_addr;
218                 t=connect(s3,(struct sockaddr *)&server,sizeof(struct sockaddr_in));
219                 if(t==-1)
220                 {
221                     perror("Connect to server failed");
222                     exit(0);
223                 }
224
225                 sprintf(response,"HTTP/1.1 200 Established\r\n\r\n");
226                 write(s2,response,strlen(response));
227
228                 if(!(pid=fork())) //Child
229                 {
230                     while(t=read(s2,request2,2000))
231                     {
232                         write(s3,request2,t);
233                         printf("CL >>>(%d)%s \n",t,host); //SOLO PER CHECK
234                     }
235                     exit(0);
236                 }
237                 else //Parent
238                 {
239                     while(t=read(s3,response2,2000))
240                     {
241                         write(s2,response2,t);
242                         printf("CL <<<(%d)%s \n",t,host);
243                     }
244
245                     kill(pid,15);
246                     shutdown(s3,SHUT_RDWR);
247                     close(s3);
248                 }
249             }
250     }
251
252         shutdown(s2,SHUT_RDWR);
253     close(s2);
254     exit(0);
255   }
256 }
```

D.2.3.4 Filter of Content-Type of response

```
#include <sys/types.h>          /* See NOTES */
#include <signal.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netinet/ip.h> /* superset of previous */
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <netdb.h>

#include "net_utility.h"
#define NUM_BLOCKED_IP 4

struct hostent * he;
struct sockaddr_in local,remote,server;
char request[10000],response[2000],request2[2000],response2[2000];
char * method, *path, *version, *host, *scheme, *resource,*port;
char blocked_IPs[NUM_BLOCKED_IP][4] = {{192,168,1,81},
                                       {192,168,1,210},
                                       {192,168, 1,14},
                                       {192,165,22,1}};

struct headers {
    char *n;
    char *v;
}h[30];

int main()
{
    FILE *f;
    char *type, *sub_type;
    char command[100], c;
    int i,s,t,s2,s3,n,len,yes=1,j,k,pid,size, block=0;

    s = socket(AF_INET, SOCK_STREAM, 0);
    if ( s == -1)
    {
        perror("Socket Failed\n");
        return 1;
    }

    local.sin_family=AF_INET;
    local.sin_port = htons(8080);
    local.sin_addr.s_addr = 0;
    setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int));
    t = bind(s,(struct sockaddr *) &local, sizeof(struct sockaddr_in));
    if ( t == -1)
    {
        perror("Bind Failed \n");
        return 1;
    }

    t = listen(s,10);
    if ( t == -1)
    {
        perror("Listen Failed \n");
        return 1;
    }

    while( 1 )
    {
        f = NULL;
        remote.sin_family=AF_INET;
        len = sizeof(struct sockaddr_in);

        s2 = accept(s,(struct sockaddr *) &remote, &len);
```

```
69          for(i=0; i<NUM_BLOCKED_IP; i++)
70          {
71              if(block = ((*(unsigned int*) blocked_IPs[i]) == remote.sin_addr.s_addr))
72                  break;
73          }
74          printf("remote: ");
75          for(i=0; i<3; i++)
76              printf("%u.", ((unsigned char*) &remote.sin_addr.s_addr)[i]);
77          printf("%u       block: %d\n", ((unsigned char*) &remote.sin_addr.s_addr)[i], block)
    ;
78
79          if(fork()) continue; //<< MULTI PROCESS HADLING
80          if (s2 == -1)
81          {
82              perror("Accept Failed\n");
83              return 1;
84          }
85
86          // <---- ADDED HEADER PARSER
87          j=0;k=0;
88          h[k].n = request;
89          while(read(s2,request+j,1))
90          {
91              if((request[j]=='\n') && (request[j-1]=='\r'))
92              {
93                  request[j-1]=0;
94
95                  if(h[k].n[0]==0)
96                      break;
97
98                  h[++k].n=request+j+1;
99              }
100
101             if(request[j]==':' && (h[k].v==0) && k!=0)
102             {
103                 request[j]=0;
104                 h[k].v=request+j+1;
105             }
106
107             j++;
108         }
109
110     method = request;
111     for(i=0;(i<2000) && (request[i]!=' ');i++); request[i]=0;
112     path = request+i+1;
113     for(    ;(i<2000) && (request[i]!=' ');i++); request[i]=0;
114     version = request+i+1;
115     printf("\n%s%s %s %s%s\n", BOLD_GREEN, method, path, version, DEFAULT);
116
117         if(!strcmp("GET",method))
118         {
119             //  http://www.google.com/path
120       scheme=path;
121       for(i=0;path[i]!=':';i++); path[i]=0;
122       host=path+i+3;
123       for(i=i+3;path[i]!='/';i++); path[i]=0;
124       resource=path+i+1;
125             printf("Scheme=%s, host=%s, resource = %s\n", scheme,host,resource);
126
127             he = gethostbyname(host);
128       if (he == NULL)
129             {
130                 printf("Gethostbyname Failed\n");
131                 return 1;
132             }
133
134       printf("Server address = %u.%u.%u.%u\n", (unsigned char ) he->h_addr[0],(unsigned
    char ) he->h_addr[1],(unsigned char ) he->h_addr[2],(unsigned char ) he->h_addr[3]);
135       s3=socket(AF_INET,SOCK_STREAM,0);
```

```
136         if(s3==-1)
137                 {
138                     perror("Socket to server failed");
139                     return 1;
140                 }
141 
142                 server.sin_family=AF_INET;
143         server.sin_port=htons(80);
144         server.sin_addr.s_addr=*(unsigned int*) he->h_addr;
145         t=connect(s3,(struct sockaddr *)&server,sizeof(struct sockaddr_in));
146                 if(t==-1)
147                 {
148                     perror("Connect to server failed");
149                     return 1;
150                 }
151 
152         sprintf(request2,"GET /%s HTTP/1.1\r\nHost:%s\r\nConnection:close\r\n\r\n",resource,
    host);
153                 write(s3,request2,strlen(request2));
154 
155                 memset(h, 0, 30*sizeof(struct headers));
156 
157                 j=0;k=0;
158                 h[k].n = response;
159                 while((t=read(s3,response+j,1))>0)
160                 {
161                     if((response[j]=='\n') && (response[j-1]=='\r'))
162                     {
163                         response[j-1]=0;
164 
165                         if(h[k].n[0]==0)
166                             break;
167 
168                         h[++k].n=response+j+1;
169                     }
170 
171                     if(response[j]==':' && (h[k].v==0) && k!=0)
172                     {
173                         response[j]=0;
174                         h[k].v=response+j+1;
175                     }
176 
177                     j++;
178                 }
179 
180                 if(t==-1)
181                 {
182                     perror("Error on message");
183                     exit(1);
184                 }
185 
186                 if(block)
187                 {
188                     for(i=1; h[i].n[0]; i++)
189                     {
190                         if(!strcmp(h[i].n, "Content-Type"))
191                         {
192                             type = h[i].v;
193                             for(j=0; h[i].v[j]!='/'; j++);
194                             h[i].v[j]=0;
195 
196                             printf("%s%15s%s/", BOLD_YELLOW, type, DEFAULT);
197 
198                             if(!strcmp(type, " text"))
199                             {
200                                 block=0;
201                                 h[i].v[j]='/';
202                             }
203 
204                             sub_type = h[i].v + j +1;
```

```
205                             int size_sub=strlen(sub_type);
206                             for(j=j+1; j<size_sub && h[i].v[j]!=';'; j++);
207                             h[i].v[j]=0;
208
209                             printf("%s%-15s%s", BOLD_CYAN, sub_type, DEFAULT);
210
211                             if(block && !strcmp(sub_type, "html"))
212                                 block = 0;
213
214                             if(j<size_sub)
215                                 h[i].v[j]=';';
216
217                             break;
218                         }
219                     }
220                 }
221
222                 if(block)
223                 {
224                     sprintf(response2, "HTTP/1.1 401 Unauthorized\r\n\r\n");
225                     write(s2, response2, strlen(response2));
226                     printf("       %s%s%s", BOLD_RED, response2, DEFAULT);
227                 }
228                 else
229                 {
230                     sprintf(response2, "%s\r\n", h[0].n);
231                     write(s2, response2, strlen(response2));
232                     printf("       %s%s%s", BOLD_BLUE, response2, DEFAULT);
233
234                     for(i=1; h[i].n[0]; i++)
235                     {
236                         sprintf(response2, "%s:%s\r\n", h[i].n, h[i].v);
237                         write(s2, response2, strlen(response2));
238                     }
239
240                     sprintf(response2, "\r\n");
241                     write(s2, response2, 2);
242
243                     while((t=read(s3, response2, 2000))>0)
244                         write(s2, response2, t);
245
246                     if(t==-1)
247                     {
248                         perror("[PROXY ERROR] Reading server response");
249                         exit(1);
250                     }
251                 }
252
253                 shutdown(s3,SHUT_RDWR);
254                 close(s3);
255         }
256         else if(!strcmp("CONNECT",method))
257             {
258           host=path;
259           for(i=0;path[i]!=':';i++); path[i]=0;
260           port=path+i+1;
261           printf("host:%s, port:%s\n",host,port);
262           printf("Connect skipped ...\n");
263           he = gethostbyname(host);
264           if (he == NULL)
265                 {
266                     printf("Gethostbyname Failed\n");
267                     return 1;
268                 }
269
270           printf("Connecting to address = %u.%u.%u.%u\n", (unsigned char ) he->h_addr[0],(
      unsigned char ) he->h_addr[1],(unsigned char ) he->h_addr[2],(unsigned char ) he->h_addr
      [3]);
271           s3=socket(AF_INET,SOCK_STREAM,0);
272           if(s3==-1)
```

```
273             {
274                  perror("Socket to server failed");
275                  return 1;
276             }
277
278       server.sin_family=AF_INET;
279       server.sin_port=htons((unsigned short)atoi(port));
280       server.sin_addr.s_addr=*(unsigned int*) he->h_addr;
281       t=connect(s3,(struct sockaddr *)&server,sizeof(struct sockaddr_in));
282             if(t==-1)
283             {
284                  perror("Connect to server failed");
285                  exit(0);
286             }
287
288             sprintf(response,"HTTP/1.1 200 Established\r\n\r\n");
289       write(s2,response,strlen(response));
290
291             if(!(pid=fork())) //Child
292             {
293         while(t=read(s2,request2,2000))
294                  {
295                      write(s3,request2,t);
296           printf("CL >>>(%d)%s \n",t,host);
297                  }
298         exit(0);
299             }
300       else //Parent
301             {
302         while(t=read(s3,response2,2000))
303                  {
304           write(s2,response2,t);
305           printf("CL <<<(%d)%s \n",t,host);
306                  }
307
308                  kill(pid,15);
309         shutdown(s3,SHUT_RDWR);
310         close(s3);
311             }
312     }
313     shutdown(s2,SHUT_RDWR);
314     close(s2);
315     exit(0);
316   }
317 }
```

D.2.3.5 Whitelist

```
#include <sys/types.h>           /* See NOTES */
#include <signal.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netinet/ip.h> /* superset of previous */
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <netdb.h>

#include "net_utility.h"
#define SIZE_WHITE_LIST 3

struct hostent * he;
struct sockaddr_in local,remote,server;
char request[10000],response[2000],request2[2000],response2[2000];
char * method, *path, *version, *host, *scheme, *resource,*port;
char white_list[SIZE_WHITE_LIST][20] = {"www.google.com",
                                        "www.radioamatori.it",
                                        "www.youtube.com"};

struct headers {
    char *n;
    char *v;
}h[30];

int main()
{
    FILE *f;
    char *type, *sub_type;
    char command[100], c;
    int i,s,t,s2,s3,n,len,yes=1,j,k,pid,size, block=1;

    s = socket(AF_INET, SOCK_STREAM, 0);
    if ( s == -1)
    {
        perror("Socket Failed\n");
        return 1;
    }

    local.sin_family=AF_INET;
    local.sin_port = htons(8080);
    local.sin_addr.s_addr = 0;
    setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int));
    t = bind(s,(struct sockaddr *) &local, sizeof(struct sockaddr_in));
    if ( t == -1)
    {
        perror("Bind Failed \n");
        return 1;
    }

    t = listen(s,10);
    if ( t == -1)
    {
        perror("Listen Failed \n");
        return 1;
    }

    while( 1 )
    {
        f = NULL;
        remote.sin_family=AF_INET;
        len = sizeof(struct sockaddr_in);

        s2 = accept(s,(struct sockaddr *) &remote, &len);

```

```
69          if(fork()) continue; //<< MULTI PROCESS HADLING
70
71          if (s2 == -1)
72          {
73              perror("Accept Failed\n");
74              return 1;
75          }
76
77          // <---- ADDED HEADER PARSER
78          j=0;k=0;
79          h[k].n = request;
80          while(read(s2,request+j,1))
81          {
82              if((request[j]=='\n') && (request[j-1]=='\r'))
83              {
84                  request[j-1]=0;
85
86                  if(h[k].n[0]==0)
87                      break;
88
89                  h[++k].n=request+j+1;
90              }
91
92              if(request[j]==':' && (h[k].v==0) && k!=0)
93              {
94                  request[j]=0;
95                  h[k].v=request+j+1;
96              }
97
98              j++;
99          }
100
101     printf("%s",request);
102     method = request;
103     for(i=0;(i<2000) && (request[i]!=' ');i++); request[i]=0;
104     path = request+i+1;
105     for(   ;(i<2000) && (request[i]!=' ');i++); request[i]=0;
106     version = request+i+1;
107         printf("Method = %s, path = %s, version = %s\n",method,path,version);
108
109         if(!strcmp("GET",method))
110         {
111             // http://www.google.com/path
112       scheme=path;
113       for(i=0;path[i]!=':';i++); path[i]=0;
114       host=path+i+3;
115       for(i=i+3;path[i]!='/';i++); path[i]=0;
116       resource=path+i+1;
117       printf("Scheme=%s, host=%s, resource = %s\n", scheme,host,resource);
118
119             for(i=0; i<SIZE_WHITE_LIST; i++)
120             {
121                 if(!strcmp(host, white_list[i]))
122                 {
123                     block=0;
124                     break;
125                 }
126             }
127
128             if(block)
129             {
130                 sprintf(response2, "HTTP/1.1 401 Unauthorized\r\n\r\n");
131                 write(s2, response2, strlen(response2));
132             }
133             else
134             {
135                 he = gethostbyname(host);
136                 if (he == NULL)
137                 {
138                     printf("Gethostbyname Failed\n");
```

```
139                     return 1;
140                 }
141 
142                 printf("Server address = %u.%u.%u.%u\n", (unsigned char ) he->h_addr[0],(
    unsigned char ) he->h_addr[1],(unsigned char ) he->h_addr[2],(unsigned char ) he->h_addr
    [3]);
143                 s3=socket(AF_INET,SOCK_STREAM,0);
144                 if(s3==-1)
145                 {
146                     perror("Socket to server failed");
147                     return 1;
148                 }
149 
150                 server.sin_family=AF_INET;
151                 server.sin_port=htons(80);
152                 server.sin_addr.s_addr=*(unsigned int*) he->h_addr;
153                 t=connect(s3,(struct sockaddr *)&server,sizeof(struct sockaddr_in));
154                 if(t==-1)
155                 {
156                     perror("Connect to server failed");
157                     return 1;
158                 }
159 
160           sprintf(request2,"GET /%s HTTP/1.1\r\nHost:%s\r\nConnection:close\r\n\r\n",
    resource,host);
161                 write(s3,request2,strlen(request2));
162 
163                 while((t=read(s3, response2, 2000))>0)
164                     write(s2, response2, t);
165 
166                 if(t==-1)
167                 {
168                     perror("[PROXY ERROR] Reading server response");
169                     exit(1);
170                 }
171 
172                 shutdown(s3,SHUT_RDWR);
173                 close(s3);
174             }
175     }
176     else if(!strcmp("CONNECT",method))
177         {
178             // www.google.com:400
179       host=path;
180       for(i=0;path[i]!=':';i++); path[i]=0;
181       port=path+i+1;
182       printf("host:%s, port:%s\n",host,port);
183 
184             for(i=0; i<SIZE_WHITE_LIST; i++)
185             {
186                 if(!strcmp(host, white_list[i]))
187                 {
188                     block=0;
189                     break;
190                 }
191             }
192 
193             if(block)
194             {
195                 sprintf(response2, "HTTP/1.1 401 Unauthorized\r\n\r\n");
196                 write(s2, response2, strlen(response2));
197             }
198             else
199             {
200                 he = gethostbyname(host);
201                 if (he == NULL)
202                 {
203                     printf("Gethostbyname Failed\n");
204                     return 1;
205                 }
```

```
206
207                 printf("Connecting to address = %u.%u.%u.%u\n", (unsigned char ) he->h_addr
    [0],(unsigned char ) he->h_addr[1],(unsigned char ) he->h_addr[2],(unsigned char ) he->
    h_addr[3]);
208                 s3=socket(AF_INET,SOCK_STREAM,0);
209                 if(s3==-1)
210                 {
211                     perror("Socket to server failed");
212                     return 1;
213                 }
214
215                 server.sin_family=AF_INET;
216                 server.sin_port=htons((unsigned short)atoi(port));
217                 server.sin_addr.s_addr=*(unsigned int*) he->h_addr;
218                 t=connect(s3,(struct sockaddr *)&server,sizeof(struct sockaddr_in));
219                 if(t==-1)
220                 {
221                     perror("Connect to server failed");
222                     exit(0);
223                 }
224
225                 sprintf(response,"HTTP/1.1 200 Established\r\n\r\n");
226                 write(s2,response,strlen(response));
227
228                 if(!(pid=fork())) //Child
229                 {
230                     while(t=read(s2,request2,2000))
231                     {
232                         write(s3,request2,t);
233                         printf("CL >>>(%d)%s \n",t,host); //SOLO PER CHECK
234                     }
235                     exit(0);
236                 }
237                 else //Parent
238                 {
239                     while(t=read(s3,response2,2000))
240                     {
241                         write(s2,response2,t);
242                         printf("CL <<<(%d)%s \n",t,host);
243                     }
244
245                     kill(pid,15);
246                     shutdown(s3,SHUT_RDWR);
247                     close(s3);
248                 }
249             }
250     }
251
252         shutdown(s2,SHUT_RDWR);
253     close(s2);
254     exit(0);
255   }
256 }
```

D.2.4 Web Server

D.2.4.1 Standard version with management of functions

```c
#include "ws.h"
#include "net_utility.h"

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <signal.h>

struct sockaddr_in local, remote;

int main()
{
    char request[2000], response[2000];
    char *method, *path, *version;
    int sd, sd2;
    int t;
    socklen_t len;
    int yes = 1;
    FILE *f;

    signal(SIGINT, endDaemon);

    //Initialization of TCP socket for IPv4 protocol
    sd = socket(AF_INET, SOCK_STREAM, 0);
    control(sd, "Socket failed\n");

    //Bind the server to a specific port
    local.sin_family=AF_INET;
    local.sin_port = htons(8080); //we need to use a port not in use
    local.sin_addr.s_addr = 0; //By default, it

    //Reuse the same IP already bind to other program
    setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));
    t = bind(sd, (struct sockaddr*) &local, sizeof(struct sockaddr_in));
    control(t, "Bind failed \n");

    //Queue of pending clients that want to connect
    t = listen(sd, QUEUE_MAX);
    control(t, "Listen failed \n");

    while(1)
    {
        f=NULL;
        remote.sin_family = AF_INET;
        len = sizeof(struct sockaddr_in);

        //Accept the new request and create its socket
        sd2 = accept(sd, (struct sockaddr*) &remote, &len);
        control(sd2, "Accept failed \n");

        //A child manages the single request
        if(!fork())
        {
            //Read the request of the client
            t = read(sd2, request, 1999);
            request[t]=0;

            //Parser of request line
            request_line(request, &method, &path, &version);
```

```
67              printf("Method:␣␣%s\n", method);
68              printf("Path:␣␣%s\n", path);
69              printf("Version:␣␣%s\n", version);
70
71              //Manage the response to the request
72              manage_request(method, path, version, response, &f);
73              printf("%s", response);
74              write(sd2, response, strlen(response));
75              send_body(sd2, f);
76
77              //Shutdown the socket created with the specific client
78              shutdown(sd2, SHUT_RDWR);
79              close(sd2);
80              exit(0);
81          }
82      }
83  }
84
85
86  void request_line(char* request, char** method, char** path, char** version)
87  {
88      int i;
89      *method = request;
90
91      for(i=1; request[i]!='␣'; i++);
92
93      request[i]=0;
94      *path=request+i+1;
95
96      for(; request[i]!='␣'; i++);
97
98      request[i]=0;
99      *version=request+i+1;
100
101     for(; (request[i]!='\n' || request[i-1]!='\r') ; i++);
102
103     request[i-1]=0;
104 }
105
106 void manage_request(char* method, char* path, char* version, char* response, FILE** f)
107 {
108     if(strcmp(method,"GET")) //it's not GET request
109         sprintf(response, "HTTP/1.1␣501␣Not␣Implemented\r\n\r\n");
110     /*
111      * else if((*f=fopen(path+1,"r"))==NULL) //it's GET request for a file
112         //path+1 is used to remove the / root directory
113         sprintf(response,"HTTP/1.1 404 Not Found\r\nConnection: close\r\n\r\n");
114     else
115         sprintf(response,"HTTP/1.1 200 Not Found\r\nConnection: close\r\n\r\n");
116     */
117     else
118     {
119         char file_name[40];
120         sprintf(file_name,"%s%s",ROOT_PATH,path);
121
122         if(!strncmp(path, CGI_BIN, 9))
123         {
124             int i=0;
125             char* arguments[10];
126
127             int size_path =strlen(path);
128             for(i=9; i<size_path && path[i]!='?'; i++);
129
130             printf("%d\n", i);
131             path[i]=0;
132             int j=0;
133             for(i=i+1; i<size_path && j<10; i++)
134             {
135                 if(path[i]=='=')
136                     arguments[j++]=path+i+1;
```

```
137
138                 if(path[i]=='&')
139                     path[i]=0;
140             }
141
142             char command[60];
143             sprintf(command, "cd %s ; %s", ROOT_PATH, path+9);
144
145             for(i=0; i<j; i++)
146             {
147                 int size = strlen(command);
148                 sprintf(command+size, " %s", arguments[i]);
149
150                 printf("%s", arguments[i]);
151             }
152
153             int size = strlen(command);
154             sprintf(command+size, " > %s", CGI_RESULT);
155             printf("%s\n", command);
156
157             int status = system(command);
158
159             if(status==-1)
160             {
161                 //Used to manage if a program doesn't exists
162                 sprintf(response,"HTTP/1.1 400 Not Found\r\nConnection:Close\r\n\r\n");
163                 *f=NULL;
164             }
165             else if(!status)
166             {
167                 //Useless if because the file is always created, because of pipe
    implementation
168                 if(((*f)=fopen(CGI_RESULT, "r+"))==NULL)
169                 {
170                     perror("Error with CGI");
171                 }
172                 else
173                     sprintf(response,"HTTP/1.1 200 OK\r\nConnection:Close\r\n\r\n");
174             }
175
176         }
177         else
178         {
179             printf("%s\n", file_name);
180
181             //"r+" because in linux directory are file so we need to specify
182             //also writing rights to be sure that fopen return NULL with also directory
183             if(((*f)=fopen(file_name,"r+"))==NULL) //it's GET request for a file
184                 sprintf(response,"HTTP/1.1 404 Not Found\r\nConnection:Close\r\n\r\n");
185             else
186                 sprintf(response,"HTTP/1.1 200 OK\r\nConnection:Close\r\n\r\n");
187         }
188     }
189 }
190
191 void send_body(int sd2, FILE* f)
192 {
193     char c;
194     if(f!=NULL)
195     {
196         while((c=fgetc(f))!=EOF)
197             write(sd2, &c, 1);
198
199         fclose(f);
200     }
201 }
202
203 void endDaemon(int sig)
204 {
205     FILE* f;
```

```
206 ||
207 ||      if((f=fopen(CGI_RESULT,"r+"))!=NULL)
208 ||      {
209 ||          char command[40];
210 ||          sprintf(command, "rm␣%s", CGI_RESULT);
211 ||          system(command);
212 ||      }
213 ||
214 ||      exit(0);
215 || }
```

D.2.4.2 Caching management

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

#define __USE_XOPEN
#include <time.h>

#define LINE "------------------------------------------------------------"
struct sockaddr_in local,remote;
char request[2000],response[2000];
char * method, *path, *version;

struct header{
    char* name;
    char* value;
}h[30];

int main()
{
    FILE *f;
    char command[100];
    int i,s,t,s2,n,len,c,yes=1, head;
    char* cache_date;

    s = socket(AF_INET, SOCK_STREAM, 0);
    if ( s == -1) { perror("Socket Failed\n"); return 1;}

    local.sin_family=AF_INET;
    local.sin_port = htons(8083);
    local.sin_addr.s_addr = 0;

    setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int));
    t = bind(s,(struct sockaddr *) &local, sizeof(struct sockaddr_in));
    if ( t == -1) { perror("Bind Failed \n"); return 1;}

    t = listen(s,10);
    if ( t == -1) { perror("Listen Failed \n"); return 1;}

    printf("%s\n", LINE);

    while( 1 )
    {
        remote.sin_family=AF_INET;
        len = sizeof(struct sockaddr_in);
        memset(&remote, 0, sizeof(struct sockaddr_in));

        s2 = accept(s,(struct sockaddr *) &remote, &len);
        if (s2 == -1) {perror("Accept Failed\n"); return 1;}

        if (!fork())
        {
            int keep_alive= 0;
            int is_updated = 0;

            f = NULL; // <<<<< BACO
            head=0;
            n=read(s2,request,1999);
            request[n]=0;
            method = request;
            cache_date = NULL;

```

```
69              for(i=0;(i<n) && (request[i]!=' ');i++); request[i]=0;
70              path = request+i+1;
71              for(   ;(i<n) && (request[i]!=' ');i++); request[i]=0;
72              version = request+i+1;
73              for(   ;(i<n) && (request[i]!='\r');i++); request[i]=0;
74
75              printf("%s %s %s\n", method, path, version);
76
77              i+=2;
78              int k=0;
79              h[k].name = request+i;
80              while(i<n)
81              {
82                  if(request[i]=='\n' && request[i-1]=='\r')
83                  {
84                      request[i-1]=0;
85
86                      if(h[k].name[0]==0)
87                          break;
88
89                      h[++k].name=request+i+1;
90                  }
91                  else if(request[i]==':' && h[k].value==0)
92                  {
93                      request[i]=0;
94                      h[k].value = request+i+1;
95                  }
96                  i++;
97              }
98
99              if(!strcmp(version, "HTTP/1.1"))
100                 keep_alive=1;
101             else if(!strcmp(version, "HTTP/1.0"))
102             {
103                 for(i=0; h[i].name[0]; i++)
104                 {
105                     if(!strcmp(h[i].name, "Connection") && !strcmp(h[i].value, "keep_alive"
    ))
106                         keep_alive=1;
107                     else if(!strcmp(h[i].name, "If-Modified-Since"))
108                         cache_date = h[i].value;
109
110                     printf("%s:%s\n",h[i].name, h[i].value);
111                 }
112             }
113
114             if(!strcmp("GET",method))
115             { // it is a get
116                 if(!strncmp(path,"/cgi-bin/",9))
117                 { // CGI interface
118                     sprintf(command,"%s > results.txt",path+9);
119                     printf("executing %s\n", command);
120                     system(command);
121
122                     if((f=fopen("results.txt","r"))==NULL)
123                     {
124                         printf("cgi bin error\n");
125                         return 1;
126                     }
127
128                     sprintf(response,"HTTP/1.1 200 OK\r\nConnection:close\r\n\r\n");
129                 }
130                 else if((f=fopen(path+1,"r"))==NULL)
131                     sprintf(response,"HTTP/1.1 404 Not Found\r\nConnection:close\r\n\r\n");
132                 else
133                 {
134                     if(cache_date!=NULL)
135                     {
136                         struct stat attr;
137                         struct tm tm;
```

```
138                             struct tm cache_tm;
139                             char date[30];
140                             stat(path+1, &attr);
141                             tm = *(gmtime(&attr.st_mtime));
142                             strptime(cache_date, "%a, %d %b %Y %H:%M:%S %Z", &cache_tm);
143
144                             if(timegm(&tm)>timegm(&cache_tm))
145                                 sprintf(response,"HTTP/1.1 200 OK\r\nConnection:close\r\n\r\n")
    ;
146                             else
147                             {
148                                 is_updated = 1;
149                                 sprintf(response,"HTTP/1.1 304 Not Modified\r\nConnection:close
    \r\n\r\n");
150                             }
151                         }
152                         else
153                             sprintf(response,"HTTP/1.1 200 OK\r\nConnection:close\r\n\r\n");
154                     }
155                 }
156                 else if(!strcmp("HEAD", method))
157                 {
158                     head=1;
159                     if(strncmp(path,"/cgi-bin/", 9))
160                     {
161                         if((f=fopen(path+1, "r"))!=NULL)
162                         {
163                             struct stat attr;
164                             struct tm tm;
165                             memset(&tm, 0, sizeof(tm));
166
167                             char date[30];
168                             stat(path+1, &attr);
169                             tm = *(gmtime(&attr.st_mtime));
170                             //strptime(gmtime(&attr.st_mtime), "%a %b  %d %H:%M:%S %Y", &tm);
171                             strftime(date, 30, "%a, %d %b %Y %H:%M:%S %Z", &tm);
172                             sprintf(response, "HTTP/1.1 200 OK\r\nConnection:keep-alive\r\nLast
    -Modified:%s\r\n\r\n", date);
173                         }
174                         else
175                             sprintf(response,"HTTP/1.1 404 Not Found\r\nConnection:close\r\n\r\
    n");
176                     }
177                 }
178                 else
179                     sprintf(response, "HTTP/1.1 501 Not Implemented\r\n\r\n");
180
181                 write(s2,response,strlen(response)); // HTTP response line
182
183                 if(f!=NULL)
184                 {
185                     // if present, the Entity Body
186                     if(!head && !is_updated)
187                     {
188                         while((c=fgetc(f))!=EOF)
189                             write(s2,&c,1);
190                     }
191
192                     fclose(f);
193                 }
194
195                 printf("%s\n", LINE);
196                 shutdown(s2,SHUT_RDWR);
197                 close(s2);
198                 exit(0);
199             }
200         }
201     }
```

D.2.4.3 Management of Transfer-Encoding:chunked

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netinet/ip.h> /* superset of previous */
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

struct sockaddr_in local,remote;
char request[2000],response[2000];
char * method, *path, *version;
#define SIZE_CHUNK 11

int main()
{
    FILE *f;
    char command[100];
    int i,s,t,s2,n,len,c,yes=1;
    unsigned int count;
    char response_temp[SIZE_CHUNK];

    s = socket(AF_INET, SOCK_STREAM, 0);
    if ( s == -1) { perror("Socket Failed\n"); return 1;}

    local.sin_family=AF_INET;
    local.sin_port = htons(8083);
    local.sin_addr.s_addr = 0;

    setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int));
    t = bind(s,(struct sockaddr *) &local, sizeof(struct sockaddr_in));
    if ( t == -1) { perror("Bind Failed \n"); return 1;}

    t = listen(s,10);
    if ( t == -1) { perror("Listen Failed \n"); return 1;}

    while( 1 )
    {
        f = NULL; // <<<<< BACO
        remote.sin_family=AF_INET;
        len = sizeof(struct sockaddr_in);
        s2 = accept(s,(struct sockaddr *) &remote, &len);
        if (s2 == -1) {perror("Accept Failed\n"); return 1;}

        if (!fork())
        {
            n=read(s2,request,1999);
            request[n]=0;
            printf("%s",request);
            method = request;
            for(i=0;(i<2000) && (request[i]!=' ');i++);
            request[i]=0;
            path = request+i+1;

            for(   ;(i<2000) && (request[i]!=' ');i++);
            request[i]=0;
            version = request+i+1;

            for(   ;(i<2000) && (request[i]!='\r');i++);
            request[i]=0;

            printf("Method = %s, path = %s , version = %s\n",method,path,version);

            if(strcmp("GET",method)) // it is not a GET
                sprintf(response, "HTTP/1.1 501 Not Implemented\r\n\r\n");
            else
            { // it is a get
```

```
69                  if(!strncmp(path,"/cgi-bin/",9))
70                  {
71                      // CGI interface
72                      sprintf(command,"%s > results.txt",path+9);
73                      printf("executing %s\n", command);
74                      system(command);
75
76                      if((f=fopen("results.txt","r"))==NULL)
77                      {
78                          printf("cgi bin error\n");
79                          return 1;
80                      }
81                      sprintf(response,"HTTP/1.1 200 OK\r\nTransfer-Encoding: chunked\r\n\r\n
      ");
82                  }
83                  else  if((f=fopen(path+1,"r"))==NULL)
84                      sprintf(response,"HTTP/1.1 404 Not Found\r\nConnection:Close\r\n\r\n");
85                  else
86                      sprintf(response,"HTTP/1.1 200 OK\r\nTransfer-Encoding: chunked\r\n\r\n
      ");
87              }
88
89              write(s2,response,strlen(response)); // HTTP Headers
90              if(f!=NULL)
91              {
92                  // if present, the Entity Body
93                  while(1)
94                  {
95                      count=0;
96                      int size = (rand() % (SIZE_CHUNK-1)) +1;
97
98                      while(count<size)
99                      {
100                         c=fgetc(f);
101                         if(c!=EOF)
102                         {
103                             response_temp[count]=c;
104                             count++;
105                             printf("%c",c);
106                         }
107                         else
108                             break;
109                     }
110                     response_temp[count]=0;
111
112                     sprintf(response,"%x\r\n%s\r\n", count, response_temp);
113                     write(s2, response, strlen(response));
114
115                     if(c==EOF)
116                         break;
117                 }
118
119                 sprintf(response, "0\r\n\r\n");
120                 write(s2, response, strlen(response));
121
122                 fclose(f);
123             }
124             shutdown(s2,SHUT_RDWR);
125             close(s2);
126             exit(0);
127         }
128     }
129 }
```

D.2.4.4 Management of Content-Length

```c
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netinet/ip.h> /* superset of previous */
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#define LINE "------------------------------------------------------------"
struct sockaddr_in local,remote;
char request[2000],response[2000];
char * method, *path, *version;

struct header{
    char* name;
    char* value;
}h[30];

int main()
{
    FILE *f;
    char command[100];
    int i,s,t,s2,n,len,c,yes=1, j;
    unsigned int count;
    char response_length[10];

    s = socket(AF_INET, SOCK_STREAM, 0);
    if ( s == -1) { perror("Socket Failed\n"); return 1;}

    local.sin_family=AF_INET;
    local.sin_port = htons(8083);
    local.sin_addr.s_addr = 0;

    setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int));
    t = bind(s,(struct sockaddr *) &local, sizeof(struct sockaddr_in));
    if ( t == -1) { perror("Bind Failed \n"); return 1;}

    t = listen(s,10);
    if ( t == -1) { perror("Listen Failed \n"); return 1;}

    while( 1 )
    {
        f = NULL; // <<<<< BACO
        remote.sin_family=AF_INET;
        len = sizeof(struct sockaddr_in);
        s2 = accept(s,(struct sockaddr *) &remote, &len);
        if (s2 == -1) {perror("Accept Failed\n"); return 1;}

        if (!fork())
        {
            n=read(s2,request,1999);
            request[n]=0;
            method = request;
            for(i=0;(i<2000) && (request[i]!=' ');i++);
            request[i]=0;
            path = request+i+1;

            for(   ;(i<2000) && (request[i]!=' ');i++);
            request[i]=0;
            version = request+i+1;

            for(   ;(i<2000) && (request[i]!='\r');i++);
            request[i]=0;

            printf("%s\nMethod = %s, path = %s , version = %s\n",LINE,method,path,version);
```

```
68
69              i+=2;
70              j=0;
71              h[j].name = request+i;
72
73              while(i<n)
74              {
75                   if(request[i]=='\r' && request[i+1]=='\n')
76                   {
77                        request[i]=0;
78                        h[++j].name=request+i+2;
79                        i++;
80                   }
81                   else if(request[i]==':' && h[j].value==0)
82                   {
83                        request[i]=0;
84                        h[j].value=request+i+1;
85                   }
86
87                   i++;
88              }
89
90
91              for(i=0; h[i].name[0]; i++)
92              {
93                   printf("[%s] %s\n", h[i].name, h[i].value);
94
95                   if((!strcmp(h[i].name, "Connection") && !strcmp(h[i].value, "keep-alive"))
96                      || !strcmp(version, "HTTP/1.1"))
97                        break;
98              }
99
100             printf("%s\n", LINE);
101
102             if(!h[i].name[0] && !strcmp(version, "HTTP/1.0"))
103                  sprintf(response, "%s 400 Bad Request\r\n\r\n", version);
104             else if(strcmp("GET",method)) // it is not a GET
105                  sprintf(response, "%s 501 Not Implemented\r\n\r\n", version);
106             else
107             { // it is a get
108                  if(!strncmp(path,"/cgi-bin/",9))
109                  {
110                       // CGI interface
111                       sprintf(command,"%s > results.txt",path+9);
112                       printf("executing %s\n", command);
113                       system(command);
114
115                       if((f=fopen("results.txt","r"))==NULL)
116                       {
117                            printf("cgi bin error\n");
118                            return 1;
119                       }
120                       sprintf(response,"%s 200 OK\r\nContent-Length", version);
121                  }
122                  else  if((f=fopen(path+1,"r"))==NULL)
123                       sprintf(response,"%s 404 Not Found\r\nConnection:Close\r\n\r\n",
    version);
124                  else
125                       sprintf(response,"%s 200 OK\r\nContent-Length:", version);
126             }
127
128             write(s2,response,strlen(response)); // HTTP Headers
129
130             if(f!=NULL)
131             {
132                  // if present, the Entity Body
133                  count=0;
134                  while((c=fgetc(f))!=EOF)
135                  {
136                       sprintf(response+count, "%c", c);
```

```
137                         count++;
138                     }
139
140                     sprintf(response_length, "%d\r\n\r\n", count);
141                     write(s2, response_length, strlen(response_length));
142                     write(s2, response, count);
143
144                     fclose(f);
145                 }
146
147                 shutdown(s2,SHUT_RDWR);
148                 close(s2);
149                 exit(0);
150             }
151     }
152 }
```

D.2.4.5 Reflect of request with additional info

```c
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netinet/ip.h> /* superset of previous */
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

struct sockaddr_in local,remote;
char request[2000],response[2000];
char * method, *path, *version;
int main()
{
    FILE *f;
    char command[100];
    int i,s,t,s2,n,len,c,yes=1, reflect=0;

    s = socket(AF_INET, SOCK_STREAM, 0);
    if ( s == -1) { perror("Socket Failed\n"); return 1;}
    local.sin_family=AF_INET;
    local.sin_port = htons(8083);
    local.sin_addr.s_addr = 0;

    setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int));
    t = bind(s,(struct sockaddr *) &local, sizeof(struct sockaddr_in));
    if ( t == -1) { perror("Bind Failed \n"); return 1;}

    t = listen(s,10);
    if ( t == -1) { perror("Listen Failed \n"); return 1;}

    while( 1 )
    {
        f = NULL; // <<<<< BACO
        remote.sin_family=AF_INET;
        len = sizeof(struct sockaddr_in);
        s2 = accept(s,(struct sockaddr *) &remote, &len);

        if (s2 == -1) {perror("Accept Failed\n"); return 1;}

        if (!fork())
        {
            n=read(s2,request,1999);
            request[n]=0;
            printf("%s",request);
            method = request;

            for(i=0;(i<2000) && (request[i]!=' ');i++);
            request[i]=0;
            path = request+i+1;

            for(   ;(i<2000) && (request[i]!=' ');i++);
            request[i]=0;
            version = request+i+1;

            for(   ;(i<2000) && (request[i]!='\r');i++);
            request[i]=0;
            printf("Method = %s, path = %s , version = %s\n",method,path,version);

            if(strcmp("GET",method)) // it is not a GET
                sprintf(response, "HTTP/1.1 501 Not Implemented\r\n\r\n");
            else
            { // it is a get
                if(!strncmp(path,"/cgi-bin/",9))
                { // CGI interface
                    sprintf(command,"%s > results.txt",path+9);
                    printf("executing %s\n", command);
```

```
69                      system(command);
70                      if((f=fopen("results.txt","r"))==NULL)
71                      {
72                          printf("cgi bin error\n");
73                          return 1;
74                      }
75
76                      sprintf(response,"HTTP/1.1 200 OK\r\nConnection: close\r\n\r\n");
77                  }
78                  else if(!strncmp(path,"/reflect", 8))
79                  {
80                      sprintf(response,"HTTP/1.1 200 OK\r\nConnection: close\r\n\r\n");
81                      reflect=1;
82                  }
83                  else  if((f=fopen(path+1,"r"))==NULL)
84                      sprintf(response,"HTTP/1.1 404 Not Found\r\nConnection: close\r\n\r\n")
    ;
85                  else
86                      sprintf(response,"HTTP/1.1 200 OK\r\nConnection: close\r\n\r\n");
87              }
88
89              write(s2, response, strlen(response)); // HTTP Headers
90              if(f!=NULL)
91              { // if present, the Entity Body
92                  while((c=fgetc(f))!=EOF)
93                      write(s2,&c,1);
94
95                  fclose(f);
96              }
97              else if(reflect)
98              {
99                  *(path-1)=' ';
100                 *(version-1)=' ';
101                 request[i]='\r';
102                 write(s2, request, 1999);
103                 unsigned char* ip_addr = (unsigned char*) &(remote.sin_addr.s_addr);
104                 sprintf(response, "\r\n%u.%u.%u.%u\r\n%d\r\n", ip_addr[0], ip_addr[1],
    ip_addr[2], ip_addr[3], ntohs(remote.sin_port));
105                 write(s2,response, strlen(response));
106             }
107
108             shutdown(s2,SHUT_RDWR);
109             close(s2);
110             exit(0);
111         }
112     }
113 }
```

D.3 base64

```
#include "base64.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char** argv)
{
    char* input;

    int code=-1;
    int input_file=0;
    FILE *f_in, *f_out;

    if(argc<2)
    {
        perror("You need to specify -d or -e\n");
        return 1;
    }

    int i=1;
    for(; i<argc; i++)
    {
        if(!strcmp(argv[i],"-d"))
        {
            if(code!=-1)
            {
                perror("Too many arguments");
                return 1;
            }

            code = DECODE;
            continue;
        }

        else if(!strcmp(argv[i],"-e"))
        {
            if(code!=-1)
            {
                perror("Too many arguments");
                return 1;
            }

            code = ENCODE;
            continue;
        }

        else
        {
            if((f_in=fopen(argv[i], "r+"))==NULL)
            {
                printf("Invalid argument\n");
                return 1;
            }
            else
            {
                input_file=1;
                continue;
            }
        }
    }

    input = malloc(sizeof(char)*LINE_SIZE);
    char *output;

    if(!input_file)
    {
        fgets(input, LINE_SIZE, stdin);
        int length = strlen(input);
```

```
69          input[length-1]=0; //remove \n
70
71          base64(input, &output, code);
72
73          printf("\n\n%s\n",LINE);
74          printf("%s", output);
75          printf("\n%s\n\n", LINE);
76      }
77      else
78      {
79          f_out=fopen(OUTPUT, "w");
80          while(fgets(input, LINE_SIZE, f_in)!=NULL)
81          {
82              if(code==DECODE)
83              {
84                  int length = strlen(input);
85                  input[length-1]=0; //remove \n
86              }
87
88              base64(input, &output, code);
89
90              fprintf(f_out, "%s", output);
91          }
92      }
93
94      free(output);
95      return 0;
96  }
97
98  void base64(char* input, char** output, int code)
99  {
100     switch(code)
101     {
102         case ENCODE:
103             encode(input, output);
104             break;
105
106         case DECODE:
107             decode(input, output);
108             break;
109     }
110 }
111
112 void encode(char* input, char** output)
113 {
114     int length_in = strlen(input);
115     int length_out;
116     int i=0;
117     int k=0;
118     unsigned int num = 0;
119     char* p = (char*) (&num);
120     unsigned int mask = 0;
121
122     length_out = (length_in%3!=0)? ((length_in/3)*4+5) : ((length_in/3)*4+1);
123     *output = malloc(sizeof(char)*length_out);
124     printf("length_in: %d\n", length_in);
125
126     int count = length_in/3;
127     printf("count: %d", count);
128
129     for(; i<count; i++)
130     {
131         int j=0;
132
133         mask = (unsigned int) 252*256*256*256;
134
135         p[3]=input[i*3];
136         p[2]=input[i*3+1];
137         p[1]=input[i*3+2];
138
```

```
139         printf("num: %u\n", num);
140
141         for(; j<4; j++)
142         {
143             unsigned int num_base = (unsigned int) (num & mask>>(6*j));
144             (*output)[k++] = encode_symbol(num_base>>((3-j)*6+8));
145         }
146     }
147
148     num=0;
149     mask = (unsigned int) 252*256*256*256;
150     printf("k: %d", k);
151
152     switch(length_in%3)
153     {
154         case(1):
155         {
156             p[3]=input[i*3];
157             unsigned int num_base = num & (mask);
158             (*output)[k++] = encode_symbol(num_base >> ((3*6)+8));
159             num_base = num & (mask>>6);
160             (*output)[k++] = encode_symbol(num_base >> ((2*6)+8));
161             (*output)[k++]='=';
162             (*output)[k++]='=';
163             break;
164         }
165
166         case(2):
167         {
168             p[3]=input[i*3];
169             p[2]=input[i*3+1];
170             unsigned int num_base = num & (mask);
171             (*output)[k++] = encode_symbol(num_base >> ((3*6)+8));
172             num_base = num & (mask>>6);
173             (*output)[k++] = encode_symbol(num_base >> ((2*6)+8));
174             num_base = num & (mask>>2*6);
175             (*output)[k++] = encode_symbol(num_base >> ((1*6)+8));
176             (*output)[k++]='=';
177
178             break;
179         }
180     }
181
182     (*output)[k]=0;
183 }
184
185 void decode(char* input, char** output)
186 {
187     int length_in = strlen(input);
188     int length_out;
189
190     if(length_in%4!=0)
191     {
192         perror("No base64 encoded string\n");
193         exit(1);
194     }
195
196     if(input[length_in-2]=='=')
197     {
198         if(input[length_in-3]=='=')
199             length_out = (length_in/4)*3-1;
200         else
201             length_out = (length_in/4)*3;
202     }
203     else
204         length_out = (length_in/4)*3+1;
205
206     *output = malloc(sizeof(char)*length_out);
207
208
```

```
209     int i=0;
210     int k=0;
211
212     for(; i<(length_in/4); i++)
213     {
214         int j=0;
215         unsigned int num_base=0;
216         char* p = (char*) &num_base;
217
218         for(; j<4; j++)
219         {
220             unsigned int num = decode_symbol(input[i*4+j]);
221
222             printf("   %d\n", num);
223             num_base = num_base | (num<<((3-j)*6));
224         }
225
226         int min=0;
227
228         if(i==(length_in/4-1) && length_out==((length_in/4)*3-1))
229             min = 2;
230
231         if(i==(length_in/4-1) && length_out==((length_in/4)*3))
232             min=1;
233
234         for(j=2; j>=min; j--)
235            (*output)[k++]=p[j];
236
237     }
238
239     (*output)[k]=0;
240 }
241
242 char encode_symbol(unsigned int num_symbol)
243 {
244     char base64_sym;
245
246     printf("num: %d\n", num_symbol);
247     switch(num_symbol)
248     {
249         case 0 ... 25:
250             base64_sym = 'A'+ (char) num_symbol;
251             break;
252
253         case 26 ... 51:
254             base64_sym = 'a'+(char) (num_symbol-26);
255             break;
256
257         case 52 ... 61:
258             base64_sym = '0'+ (char) (num_symbol-52);
259             break;
260
261         case 62:
262             base64_sym = '+';
263             break;
264
265         case 63:
266             base64_sym = '/';
267             break;
268
269         default:
270             printf("Not a valid number\n");
271             exit(1);
272     }
273
274     return base64_sym;
275 }
276
277 unsigned int decode_symbol(char base64_symbol)
278 {
```

```
279     unsigned char num_symbol;
280
281     printf("%c", base64_symbol);
282
283     switch(base64_symbol)
284     {
285         case 'A' ... 'Z':
286             num_symbol = (base64_symbol-'A');
287             break;
288
289         case 'a' ... 'z':
290             num_symbol = 26 + (base64_symbol-'a');
291             break;
292
293         case '0' ... '9':
294             num_symbol = 52 + (base64_symbol-'0');
295             break;
296
297         case '+':
298             num_symbol = 62;
299             break;
300
301         case '/':
302             num_symbol = 63;
303             break;
304
305         case '=':
306             num_symbol = 0;
307             break;
308     }
309
310     return num_symbol;
311 }
```

D.4 Data Link Layer

D.4.1 Structure of packets

```
/*Host (IP address+port)*/
typedef struct
{
    unsigned char mac[6]; //MAC address of the host
    unsigned char ip[4]; //IP address of the host
}host;

/*Ethernet frame format*/
typedef struct
{
    unsigned char dst[6]; //dst MAC address
    unsigned char src[6]; //src MAC address
    unsigned short int type; //type of upper layer protocol (e.g. IP, ARP,...)
    unsigned char payload[1500]; //payload
}eth_frame;

/*ARP packet format*/
typedef struct
{
    unsigned short hw; //code for HW protocol (e.g. Ethernet)
    unsigned short protocol; //code for upper layer protocol (e.g. IP)
    unsigned char hw_len; //length of HW address (6 for MAC)
    unsigned char prot_len; // length of protocol address (4 for IP)
    unsigned short op; //operation to do (e.g. ARP request/reply, rARP request/reply, ...)
    unsigned char src_MAC[6]; //src HW address
    unsigned char src_IP[4]; //src protocol address
    unsigned char dst_MAC[6]; //dst HW address
    unsigned char dst_IP[4]; //dst protocol address
}arp_pkt;

/*IP datagram format*/
typedef struct
{
    unsigned char ver_IHL; //version (8 Bytes) = 4  +  IHL (8 Bytes) = number of 32 words
    used in header = 5
    unsigned char type_service; //type of service
    unsigned short length; // length of the entire IP datagram
    unsigned short id; //identifier of the packet
    unsigned short flag_offs; // flags (Don't fragment,...)
    unsigned char ttl; //Time to live
    unsigned char protocol; //upper layer protocol (e.g. ICMP)
    unsigned short checksum; //checksum of IP header
    unsigned int src_IP; //src IP address
    unsigned int dst_IP; //dst IP address
    unsigned char payload[1500];
}ip_datagram;

/*ICMP packet format*/
typedef struct
{
    unsigned char type; //type of ICMP packet (8=ECHO request,  0=ECHO reply)
    unsigned char code; //additional specifier of type
    unsigned short checksum; //checksum of entire ICMP packet (Header+Payload)
    unsigned short id; //identifier of the packet
    unsigned short seq; //usefull to identify packet together with id
    unsigned char payload[1500];
}icmp_pkt;
```

D.4.2 Checksum of a buffer of bytes

```
#include <arpa/inet.h>

unsigned short checksum(unsigned char* buf, int size)
{
    int i;
    unsigned int sum=0;
    unsigned short* p = (unsigned short*) buf;

    for(i=0; i<size/2; i++)
    {
        sum += htons(p[i]);

        if(sum&0x10000)
            sum = (sum&0xffff)+1;
    }

    return (unsigned short) ~sum;
}
```

D.4.3 ARP implementation

```c
#include "utility.h"
#include "arp.h"
#include <sys/socket.h>
#include <linux/if_packet.h>
#include <net/ethernet.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <net/if.h>
#include <arpa/inet.h>

void arp_resolution(int sd, host* src, host* dst, char* interface,
                    unsigned char* gateway, int verbose)
{
    unsigned char packet[PACKET_SIZE];
    struct sockaddr_ll sll;
    eth_frame *eth;
    arp_pkt *arp;
    int i;
    int found = 0;
    socklen_t len;
    int n;

    //Ethernet header
    eth = (eth_frame*) packet;

    for(i=0; i<6; i++)
        eth->dst[i]=0xff; //Broadcast request

    memcpy(eth->src, src->mac, 6);
    eth->type = htons(0x0806);

    //ARP packet
    arp = (arp_pkt *) (eth->payload);
    arp->hw = htons(0x0001);
    arp->protocol = htons(0x0800);
    arp->hw_len = 6;
    arp->prot_len = 4;
    arp->op = htons(0x0001);
    memcpy(arp->src_MAC, src->mac, 6);
    memcpy(arp->src_IP, src->ip, 4);

    for(i=0; i<6; i++)
        arp->dst_MAC[i] = 0;

    int local = ((*(unsigned int*) gateway)==0)? 1 : 0;

    if(local)
    {
        printf("        The remote host is in the same LAN\n");
        memcpy(arp->dst_IP, dst->ip, 4);
    }
    else
    {
        printf("      The remote host is outside the network\n");
        memcpy(arp->dst_IP, gateway, 4);
    }

    sll.sll_family = AF_PACKET;
    sll.sll_ifindex = if_nametoindex(interface);

    len = sizeof(sll);

    if(verbose>50)
    {
        printf("\n%s                    ARP request\n%s", BOLD_BLUE, DEFAULT);
        print_packet(packet, ETH_HEADER_SIZE+sizeof(arp_pkt), BOLD_BLUE);
    }
```

```
69
70      n = sendto(sd, packet, ETH_HEADER_SIZE+sizeof(arp_pkt), 0, (struct sockaddr*) &sll,
        sizeof(sll));
71      control(n, "ARP sendto ERROR");
72
73      while(!found)
74      {
75          int n = recvfrom(sd, packet, ETH_HEADER_SIZE+sizeof(arp_pkt), 0, (struct sockaddr*)
         &sll, &len);
76
77          control(n, "ARP recvfrom ERROR");
78
79          if(eth->type == htons(0x0806) && //it's ARP
80             arp->op == htons(0x0002) && //it's ARP reply
81             ((!memcmp(arp->src_IP, dst->ip, 4) && local) ||
82             (!memcmp(arp->src_IP, gateway, 4) && !local))) //dst of ARP request = src of ARP
         reply
83              {
84                  memcpy(dst->mac, arp->src_MAC, 6);
85
86                  if(verbose>50)
87                  {
88                      printf("\n%s                    ARP reply\n%s", BOLD_BLUE, DEFAULT);
89                      print_packet(packet, ETH_HEADER_SIZE+sizeof(arp_pkt), BOLD_BLUE);
90                  }
91
92                  found = 1;
93              }
94      }
95  }
```

D.4.4 Inverse ping

```c
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <linux/if_packet.h>
#include <net/ethernet.h> /* the L2 protocols */
#include <net/if.h>
#include <string.h>
#include "utility.h"

unsigned char mymac[6]={0x4c,0xbb,0x58,0x5f,0xb4,0xdc};
unsigned char targetmac[6];
unsigned char buffer[1500];
int s;
struct sockaddr_ll sll;

struct eth_frame
{
    unsigned char dst[6];
    unsigned char src[6];
    unsigned short type;
    unsigned char payload[1460];
};

struct arp_packet
{
    unsigned short hw;
    unsigned short proto;
    unsigned char hlen;
    unsigned char plen;
    unsigned short op;
    unsigned char srcmac[6];
    unsigned char srcip[4];
    unsigned char dstmac[6];
    unsigned char dstip[4];
};

struct ip_datagram
{
    unsigned char ver_ihl;
    unsigned char tos;
    unsigned short len;
    unsigned short id;
    unsigned short flag_offs;
    unsigned char ttl;
    unsigned char proto;
    unsigned short  checksum;
    unsigned int src;
    unsigned int dst;
    unsigned char payload[1480];
};

struct icmp_packet
{
    unsigned char type;
    unsigned char code;
    unsigned short checksum;
    unsigned int unused;
    unsigned char payload[84];
};

unsigned char packet[1500];

int main()
{
    int i,n,len ;
    unsigned char mac_addr[6];
    unsigned char ip_addr[4];
```

```
69      struct eth_frame * eth;
70      struct ip_datagram * ip;
71      struct icmp_packet * icmp;
72
73      s = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
74      if(s==-1){perror("socket failed");return 1;}
75
76      sll.sll_family=AF_PACKET;
77      sll.sll_ifindex = if_nametoindex("eth0");
78      len=sizeof(sll);
79
80      eth = (struct eth_frame *) packet;
81      ip = (struct ip_datagram *) eth->payload;
82      icmp = (struct icmp_packet *) ip->payload;
83
84      while( 1 )
85      {
86          len=sizeof(sll);
87          n=recvfrom(s,packet,1500, 0,(struct sockaddr *)&sll,&len);
88
89          if (n == -1)
90          {
91              perror("Recvfrom failed");
92              return 0;
93          }
94
95          if (eth->type == htons (0x0800)) //it is IP
96          {
97              if(ip->proto == 1) // it is ICMP
98              {
99                  if(icmp->type==8)
100                 {
101                     unsigned short checksum_IP = ip->checksum;
102                     unsigned short checksum_ICMP = icmp->checksum;
103                     ip->checksum = 0;
104                     icmp->checksum = 0;
105
106                     unsigned int ip_HEADER_length = (ip->ver_ihl & 0x0F) * 4;
107                     unsigned int ICMP_length = ntohs(ip->len) - ip_HEADER_length;
108                     printf("\n%sIP HEADER length:%s %u   %sICMP length:%s %u\n", BOLD_RED,
    DEFAULT, ip_HEADER_length, BOLD_BLUE, DEFAULT, ICMP_length);
109
110                     if((ip->checksum = htons(checksum((unsigned char*) ip, ip_HEADER_length
    )))==checksum_IP &&
111                        (icmp->checksum = htons(checksum((unsigned char*) icmp, ICMP_length)
    ))==checksum_ICMP)
112                     {
113                         memcpy(mac_addr, eth->dst, 6);
114                         memcpy(eth->dst, eth->src, 6);
115                         memcpy(eth->src, mac_addr, 6);
116
117                         memcpy(ip_addr, (unsigned char*) &(ip->dst), 4);
118                         memcpy((unsigned char*) &(ip->dst), (unsigned char*) &(ip->src), 4)
    ;
119                         memcpy((unsigned char*) &(ip->src), ip_addr, 4);
120
121                         print_packet(packet,14+ip_HEADER_length+ICMP_length, BOLD_YELLOW);
122
123                         icmp->type = 0;
124                         icmp->checksum = 0;
125                         icmp->checksum = htons(checksum((unsigned char*) icmp, ICMP_length)
    );
126
127                         for(i=0;i<sizeof(sll);i++) ((char *)&sll)[i]=0;
128
129                         sll.sll_family=AF_PACKET;
130                         sll.sll_ifindex = if_nametoindex("eth0");
131                         len=sizeof(sll);
132
133                         n=sendto(s,packet, n, 0,(struct sockaddr *)&sll,len);
```

```
134 ||
135 ||                              if (n == -1)
136 ||                              {
137 ||                                   perror("Recvfrom failed");
138 ||                                   return 0;
139 ||                              }
140 ||                         }
141 ||                    }
142 ||               }
143 ||          }
144 ||     }
145 ||     return 0;
146 || }
```

D.4.5 Ping

```
#include "ping.h"
#include "utility.h"
#include "arp.h"

int verbose = MIN_VERBOSE;
double precision = 1000.0; //s=1.0 ms=1000.0 ns=1000000.0

int main(int argc, char** argv)
{
    int sd;
    int i;
    unsigned int x;
    FILE* fd;
    char command[60];
    char* interface;
    char line[LINE_SIZE];
    unsigned char network[4];
    unsigned char gateway[4];
    unsigned char mask[4];
    char mac_file[30];
    char c;
    struct hostent* he;
    struct in_addr addr;

    host src; //me
    host dst; //remote host
    int num_pkts = DEFAULT_NUM;
    int size_pkt = DEFAULT_SIZE;

    if(argc==1)
    {
        printf("You need to specify at least destination address, type --help for info");
        exit(1);
    }
    else if(argc>=2)
    {
        if(inet_aton(argv[1], &addr)==0) //input argument is not a valid IP address
        {
            he = gethostbyname(argv[1]);

            if(he == NULL)
                control(-1, "Get IP from hostname");
            else
            {
                for(i=0; i<4; i++)
                    dst.ip[i] = (unsigned char) (he->h_addr[i]);
            }

        }
        else
        {
            unsigned char *p = (unsigned char*) &(addr.s_addr);

            for(i=0; i<4; i++)
                dst.ip[i] = p[i];
        }

        if(argc>2)
        {
            int i=2;
            for(; i<argc; i++)
            {
                if(!strncmp(argv[i], "-n", 2))
                    num_pkts = atoi(argv[++i]);
                else if(!strncmp(argv[i], "-s", 2))
                    size_pkt = atoi(argv[++i]);
                else if(!strncmp(argv[i], "-v", 2))
                    verbose  = MAX_VERBOSE;
```

```
69              }
70          }
71      }
72
73      printf("\n%s---------------Remote␣␣analysis----------------------\n%s", BOLD_RED,
        DEFAULT);
74      printf("%sDestination␣address␣=␣%s",BOLD_GREEN, DEFAULT);
75      for(i=0; i<3; i++)
76      {
77          printf("%u.", dst.ip[i]);
78      }
79      printf("%u\n", dst.ip[i]);
80
81
82      //Evaluation of Ethernet interface name
83      sprintf(command, "route␣-n␣|␣tac␣|␣head␣--lines=-2␣");
84      fd = popen(command, "r");
85
86      if(fd == NULL)
87          control(-1, "Opening␣pipe..");
88
89      while(fgets(line, LINE_SIZE, fd)!=NULL)
90      {
91          char* s = strtok(line, "␣");
92          i=0;
93
94          if(s!=NULL)
95          {
96              if (inet_aton(s, &addr)!=0)
97              {
98                  unsigned char *p = (unsigned char*) &(addr.s_addr);
99
100                 memcpy(network, p, 4);
101             }
102             i++;
103         }
104
105         while((s=strtok(NULL,"␣"))!=NULL && i<8)
106         {
107             switch(i)
108             {
109                 case ROUTE_GATEWAY_INDEX:
110                 {
111                     if (inet_aton(s, &addr)!=0)
112                     {
113                         unsigned char *p = (unsigned char*) &(addr.s_addr);
114
115                         memcpy(gateway, p, 4);
116                     }
117                     break;
118                 }
119
120                 case ROUTE_MASK_INDEX:
121                 {
122                     if (inet_aton(s, &addr)!=0)
123                     {
124                         unsigned char *p = (unsigned char*) &(addr.s_addr);
125
126                         memcpy(mask,p, 4);
127                     }
128                     break;
129                 }
130
131                 case ROUTE_INTERFACE_INDEX:
132                 {
133                     s[strlen(s)-1]=0;
134                     interface = s;
135                 }
136             }
137
```

```
138             i++;
139         }
140 
141 
142         if((*(unsigned int*) &network)==((*((unsigned int*) &(dst.ip))) & (*((unsigned int
    *) &mask))))
143         {
144             break;
145         }
146     }
147     pclose(fd);
148 
149     printf("\n");
150     printf("%sGateway: %s", BOLD_MAGENTA, DEFAULT);
151     for(i=0; i<3; i++)
152         printf("%u.", gateway[i]);
153     printf("%u\n", gateway[i]);
154 
155     printf("%sNetwork: %s", BOLD_MAGENTA, DEFAULT);
156     for(i=0; i<3; i++)
157         printf("%u.", network[i]);
158     printf("%u\n", network[i]);
159 
160     printf("%s   Mask: %s", BOLD_MAGENTA, DEFAULT);
161     for(i=0; i<3; i++)
162         printf("%u.", mask[i]);
163     printf("%u\n", mask[i]);
164 
165     //See the MAC address of eth0 looking to e.g. "/sys/class/net/eth0/address" content
166     sprintf(mac_file, MAC_DEFAULT_FILE, interface);
167     fd = fopen(mac_file, "r");
168 
169     for(i=0; i<5; i++)
170     {
171         fscanf(fd, "%x:", &x);
172         src.mac[i]=(unsigned char) x;
173     }
174 
175     fscanf(fd, "%x\n", &x);
176     src.mac[i]=(unsigned char) x;
177 
178     fclose(fd);
179 
180     printf("\n");
181 
182     printf("%sEthernet Interface:%s %s\n", BOLD_CYAN, DEFAULT, interface);
183 
184     printf("%sSource MAC address: %s", BOLD_CYAN, DEFAULT);
185     for(i=0; i<5; i++)
186         printf("%x:", src.mac[i]);
187     printf("%x\n", src.mac[i]);
188 
189 
190     //Evaluation of IPv4 address of ethernet interface in input
191     sprintf(command, "ip -4 addr show %s | grep -oP '(?<=inet\\s)\\d+(\\.\\d+){3}'",
    interface);
192     fd = popen(command, "r");
193 
194     for(i=0; i<3; i++)
195     {
196         fscanf(fd, "%u%c", &x, &c);
197         src.ip[i]=x;
198     }
199 
200     fscanf(fd, "%u", &x);
201     src.ip[i]=x;
202 
203     pclose(fd);
204 
205     printf("%sSource IP address: %s", BOLD_CYAN, DEFAULT);
```

```
206     for(i=0; i<3; i++)
207         printf("%d.", src.ip[i]);
208     printf("%d\n",src.ip[i]);
209 
210 
211     //Creation of the socket
212     sd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
213     control(sd, "Socket failed");
214 
215     //ARP resolution
216     /*
217         if(myip & mask == dstip & mask)
218             arp_resolution(sd, &dst, 0.0.0.0);
219         else
220             arp_resolution(sd, &dst, gateway);
221     */
222     printf("\n%s--------------------ARP packets-----------------------\n%s", BOLD_RED,
    DEFAULT);
223     arp_resolution(sd, &src, &dst, interface, gateway, verbose);
224     printf("%sDestination MAC address: %s", BOLD_YELLOW, DEFAULT);
225     for(i=0; i<5; i++)
226         printf("%x:", dst.mac[i]);
227     printf("%x\n", dst.mac[i]);
228 
229     //Ping application
230     printf("\n%s-----------------------------------Ping
    -----------------------------------\n%s", BOLD_RED, DEFAULT);
231     ping(sd, num_pkts, size_pkt, interface, src, dst);
232     printf("%s%s%s\n", BOLD_RED, LINE_32_BITS, DEFAULT);
233 
234     return 0;
235 }
236 
237 void ping(int sd, int num_pkts, int size_pkt, char* interface, host src, host dst)
238 {
239     int i=0;
240     int count_done = 0;
241 
242     while(i<num_pkts)
243     {
244         count_done += ping_iteration(sd, i+1, size_pkt, interface, src, dst);
245         i++;
246     }
247 
248     printf("\n %sCOMPLETED: %s %d/%d\n", BOLD_YELLOW, DEFAULT, count_done, num_pkts);
249 }
250 
251 int ping_iteration(int sd, int id_pkt, int size_pkt, char* interface, host src, host dst)
252 {
253     unsigned char packet[PACKET_SIZE];
254     struct sockaddr_ll sll;
255     eth_frame *eth;
256     ip_datagram *ip;
257     icmp_pkt *icmp;
258     int i;
259     int found = 0;
260     socklen_t len;
261     int n;
262 
263     //Ethernet header
264     eth = (eth_frame*) packet;
265     memcpy(eth->src, src.mac, 6);
266     memcpy(eth->dst, dst.mac, 6);
267     eth->type = htons(0x0800);
268 
269     //IP packet
270     ip = (ip_datagram*) (eth->payload);
271     ip->ver_IHL = 0x45;
272     ip->type_service = 0;
273     ip->length = htons(ECHO_HEADER_SIZE+size_pkt+IP_HEADER_SIZE);
```

```
274     ip->id = htons(id_pkt);
275     ip->flag_offs = htons(0);
276     ip->ttl = 128;
277     ip->protocol = 1; //ICMP
278     ip->checksum = 0;
279     memcpy((unsigned char*) &(ip->src_IP), src.ip, 4);
280     memcpy((unsigned char*) &(ip->dst_IP), dst.ip, 4);
281     ip->checksum = htons(checksum((unsigned char*) ip, IP_HEADER_SIZE)); //Checksum of ip
        header
282
283
284     //Echo request (ICMP)
285     icmp = (icmp_pkt*) (ip->payload);
286     icmp->type = 8; //ECHO request
287     icmp->code = 0;
288     icmp->checksum = htons(0);
289     icmp->id = htons(id_pkt);
290     icmp->seq = htons(1);
291
292     for(i=0; i<size_pkt; i++)
293         icmp->payload[i] = i&0xff;
294
295     //Checksum of the entire packet
296     icmp->checksum = htons(checksum((unsigned char*) icmp, ECHO_HEADER_SIZE+size_pkt));
297
298     for(i=0; i<sizeof(sll); i++)
299         ((char*) &sll)[i]=0;
300
301     sll.sll_family = AF_PACKET;
302     sll.sll_ifindex = if_nametoindex(interface);
303     len = sizeof(sll);
304
305     if(verbose>50)
306     {
307         printf("\n%s␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ECHO␣request\n%s", BOLD_BLUE, DEFAULT);
308         print_packet(packet, ETH_HEADER_SIZE+IP_HEADER_SIZE+ECHO_HEADER_SIZE+size_pkt,
        BOLD_BLUE);
309     }
310
311     n = sendto(sd, packet, ETH_HEADER_SIZE+IP_HEADER_SIZE+ECHO_HEADER_SIZE+size_pkt, 0, (
        struct sockaddr*) &sll, len);
312     control(n, "ECHO␣sendto");
313
314     time_t start = clock();
315
316     while(!found)
317     {
318         len = sizeof(sll);
319         n = recvfrom(sd, packet, PACKET_SIZE, 0, (struct sockaddr*) &sll, &len);
320         control(n, "ECHO␣recvfrom");
321
322         time_t end = clock();
323
324         if(eth->type == htons(0x0800) && //IP datagram
325            ip->protocol == 1 && //ICMP packet
326            icmp->type == 0 && //ECHO reply
327            icmp->id == htons(id_pkt))
328         {
329             if(verbose>50)
330             {
331                 printf("\n%s␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ECHO␣reply\n%s", BOLD_BLUE, DEFAULT);
332                 print_packet(packet, ETH_HEADER_SIZE+IP_HEADER_SIZE+ECHO_HEADER_SIZE+
        size_pkt, BOLD_BLUE);
333             }
334
335             found = 1;
336             double elapsed_time = ((double) (end-start)/(double) CLOCKS_PER_SEC)*precision;
337             print_ping(id_pkt, ip->ttl, size_pkt, elapsed_time);
338         }
339     }
```

```
340
341     return 1;
342 }
343
344 void print_ping(int id, int ttl, int size, double elapsed_time)
345 {
346     printf("%s[Packet␣%3d]%s␣ttl:%s␣%3d␣hops␣left␣␣␣␣␣%ssize:%s␣%3d␣bytes␣␣␣␣%selapsed_time
        :%s␣%.3lf␣",
347             BOLD_CYAN, id, MAGENTA, DEFAULT, ttl, GREEN, DEFAULT, size, YELLOW, DEFAULT,
        elapsed_time);
348
349     if(precision==1.0)
350         printf("%s\n",TIME_s);
351     else if(precision==1000.0)
352         printf("%s\n",TIME_ms);
353     else if(precision==1000000.0)
354         printf("%s\n",TIME_ns);
355 }
```

D.4.6 Record route

```c
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <linux/if_packet.h>
#include <net/ethernet.h> /* the L2 protocols */
#include <net/if.h>
#include <string.h>

#include "utility.h"
#define LINE "_______________________________"

unsigned char myip[4]={88,80,187,84};
unsigned char netmask[4]={255,255,255,0};
unsigned char mymac[6]={0xf2,0x3c,0x91,0xdb,0xc2,0x98};
unsigned char gateway[4]={88,80,187,1};

//unsigned char targetip[4]={88,80,187,50};
unsigned char targetip[4]={212,71,253,5};
unsigned char targetmac[6];
unsigned char buffer[1500];
int s;
struct sockaddr_ll sll;

int printpacket(unsigned char *b,int l){
int i;
 for(i=0;i<l;i++){
  printf("%.2x(%.3d) ",b[i],b[i]);
  if(i%4 == 3) printf("\n");
  }
  printf("\n%s\n", LINE);
}

struct eth_frame
{
    unsigned char dst[6];
    unsigned char src[6];
    unsigned short type;
    unsigned char payload[1460];
};

struct arp_packet
{
    unsigned short hw;
    unsigned short proto;
    unsigned char hlen;
    unsigned char plen;
    unsigned short op;
    unsigned char srcmac[6];
    unsigned char srcip[4];
    unsigned char dstmac[6];
    unsigned char dstip[4];
};

struct ip_datagram
{
    unsigned char ver_ihl;
    unsigned char tos;
    unsigned short len;
    unsigned short id;
    unsigned short flag_offs;
    unsigned char ttl;
    unsigned char proto;
    unsigned short checksum;
    unsigned int src;
    unsigned int dst;
    unsigned char option[40];
    unsigned char payload[1441];
```

```
69  };
70
71  int forge_ip(struct ip_datagram *ip, unsigned char * dst, int payloadlen,unsigned char
        proto)
72  {
73      ip->ver_ihl=0x4F;
74      ip->tos=0;
75      ip->len=htons(payloadlen+20+40);
76      ip->id=htons(0xABCD);
77      ip->flag_offs=htons(0);
78      ip->ttl=128;
79      ip->proto=proto;
80      ip->checksum=htons(0);
81      ip->src= *(unsigned int*)myip;
82      ip->dst= *(unsigned int*)dst;
83      ip->checksum =htons(checksum((unsigned char *)ip,60));
84  };
85
86  struct icmp_packet
87  {
88      unsigned char type;
89      unsigned char code;
90      unsigned short checksum;
91      unsigned short id;
92      unsigned short seq;
93      unsigned char payload[1400];
94  };
95
96  struct record_route
97  {
98      unsigned char type;
99      unsigned char length;
100     unsigned char pointer;
101     unsigned char route_data[37];
102 };
103
104 int forge_icmp(struct icmp_packet * icmp, int payloadsize)
105 {
106     int i;
107     icmp->type=8;
108     icmp->code=0;
109     icmp->checksum=htons(0);
110     icmp->id = htons(1);
111     icmp->seq = htons(1);
112     for(i=0;i<payloadsize;i++)icmp->payload[i]=i&0xFF;
113     icmp->checksum=htons(checksum((unsigned char*)icmp,8 + payloadsize));
114 }
115
116 int arp_resolve(unsigned char* destip, unsigned char * destmac)
117 {
118     int len,n,i;
119     unsigned char pkt[1500];
120     struct eth_frame *eth;
121     struct arp_packet *arp;
122
123     printf("\n%s\n%sARP␣REQUEST%s\n", LINE, BOLD_RED, DEFAULT);
124     eth = (struct eth_frame *) pkt;
125     arp = (struct arp_packet *) eth->payload;
126
127     for(i=0;i<6;i++)
128         eth->dst[i]=0xff;
129
130     for(i=0;i<6;i++)
131         eth->src[i]=mymac[i];
132
133     eth->type=htons(0x0806);
134
135     arp->hw=htons(1);
136     arp->proto=htons(0x0800);
137     arp->hlen=6;
```

```
138     arp->plen=4;
139     arp->op=htons(1);
140
141     for(i=0;i<6;i++)
142         arp->srcmac[i]=mymac[i];
143
144     for(i=0;i<4;i++)
145         arp->srcip[i]=myip[i];
146
147     for(i=0;i<6;i++)
148         arp->dstmac[i]=0;
149
150     for(i=0;i<4;i++)
151         arp->dstip[i]=destip[i];
152
153     sll.sll_family = AF_PACKET;
154     sll.sll_ifindex = if_nametoindex("eth0");
155     len = sizeof(sll);
156
157     n=sendto(s,pkt,14+sizeof(struct arp_packet), 0,(struct sockaddr *)&sll,len);
158     printpacket(pkt,14+sizeof(struct arp_packet));
159
160     if (n == -1)
161     {
162         perror("Recvfrom failed");
163         return 0;
164     }
165
166     while( 1 )
167     {
168         n=recvfrom(s,pkt,1500, 0,(struct sockaddr *)&sll,&len);
169
170         if (n == -1)
171         {
172             perror("Recvfrom failed");
173             return 0;
174         }
175
176         if (eth->type == htons (0x0806)) //ARP packet
177         {
178             if(arp->op == htons(2)) //ARP reply
179             {
180                 if(!memcmp(destip,arp->srcip,4)) //From my target
181                 {
182                     printf("%sARP REPLY%s\n", BOLD_RED, DEFAULT);
183                     memcpy(destmac,arp->srcmac,6);
184                     printpacket(pkt,14+sizeof(struct arp_packet));
185                     return 0;
186                 }
187             }
188         }
189     }
190 }
191
192 unsigned char packet[1500];
193
194 int main()
195 {
196     int i,n,len, num_IPs, j;
197     unsigned char dstmac[6];
198
199     struct eth_frame* eth;
200     struct ip_datagram* ip;
201     struct icmp_packet* icmp;
202     struct record_route* rr;
203
204     s = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
205     if(s==-1){perror("socket failed");return 1;}
206
207     /**** HOST ROUTING ****/
```

```
208     if( (*(unsigned int*)&myip) & (*(unsigned int*)&netmask) ==
209         (*(unsigned int*)&targetip) & (*(unsigned int*)&netmask))
210         arp_resolve(targetip,dstmac);
211     else
212         arp_resolve(gateway,dstmac);
213
214     printf("%sdestmac: %s", BOLD_BLUE, DEFAULT);
215     printpacket(dstmac,6);
216
217     eth = (struct eth_frame *) packet;
218     ip = (struct ip_datagram *) eth->payload;
219     rr = (struct record_route*) ip->option;
220     icmp = (struct icmp_packet *) ip->payload;
221
222     rr->type = 7;
223     rr->length = 40;
224     rr->pointer = 4;
225
226     for(i=0;i<6;i++) eth->dst[i]=dstmac[i];
227     for(i=0;i<6;i++) eth->src[i]=mymac[i];
228     eth->type=htons(0x0800);
229     forge_icmp(icmp, 20);
230     forge_ip(ip,targetip, 20+8, 1);
231
232     printf("%sECHO REQUEST%s\n", BOLD_RED, DEFAULT);
233     printpacket(packet,14+60+8+20);
234
235     for(i=0;i<sizeof(sll);i++) ((char *)&sll)[i]=0;
236
237     sll.sll_family=AF_PACKET;
238     sll.sll_ifindex = if_nametoindex("eth0");
239     len=sizeof(sll);
240     n=sendto(s,packet,14+60+8+20, 0,(struct sockaddr *)&sll,len);
241
242     if (n == -1)
243     {
244         perror("Recvfrom failed");
245         return 0;
246     }
247
248     while( 1 )
249     {
250         len=sizeof(sll);
251         n=recvfrom(s,packet,1500, 0,(struct sockaddr *)&sll,&len);
252         if (n == -1) {perror("Recvfrom failed"); return 0;}
253
254         if (eth->type == htons (0x0800)) //IP datagram
255         {
256             if(ip->proto == 1) //ICMP packet
257             {
258                 if(icmp->type==0) //ECHO reply
259                 {
260                     printf("%sECHO REPLY%s\n", BOLD_RED, DEFAULT);
261                     if(rr->type==7)
262                     {
263                         printpacket(packet,14+60+8+20);
264                         num_IPs=(rr->pointer-4)/4;
265
266                         printf("%sRoute%s\n", BOLD_BLUE, DEFAULT);
267
268                         for(j=0; j<num_IPs; j++)
269                         {
270                             printf("%s%d:%s", BOLD_YELLOW, j+1, DEFAULT);
271                             for(i=0; i<3; i++)
272                                 printf("%u.",rr->route_data[j*4+i]);
273
274                             printf("%u\n",rr->route_data[j*4+i]);
275                         }
276
277                         break;
```

```
278                         }
279                     }
280                     else if(icmp->type==12) //Wrong IP option format
281                     {
282                         printf("%sPROBLEM%s", BOLD_RED, DEFAULT);
283                         printpacket(packet, n);
284                         break;
285                     }
286                 }
287             }
288         }
289 
290         printf("%s\n\n", LINE);
291         return 0;
292 }
```

D.4.7 Split ping

```c
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <linux/if_packet.h>
#include <net/ethernet.h> /* the L2 protocols */
#include <net/if.h>
#include <string.h>
#include <stdlib.h>

#include "utility.h"

unsigned char myip[4]={192, 168, 1, 210};
unsigned char netmask[4]={255,255,255,0};
unsigned char mymac[6]={0x4c,0xbb,0x58,0x5f,0xb4,0xdc};
unsigned char gateway[4]={192,168,1,1};

//unsigned char targetip[4]={88,80,187,50};
unsigned char targetip[4]={147,162,2,100};
unsigned char targetmac[6];
unsigned char buffer[1500];
int s;
struct sockaddr_ll sll;

struct eth_frame
{
    unsigned char dst[6];
    unsigned char src[6];
    unsigned short type;
    unsigned char payload[1460];
};

struct arp_packet
{
    unsigned short hw;
    unsigned short proto;
    unsigned char hlen;
    unsigned char plen;
    unsigned short op;
    unsigned char srcmac[6];
    unsigned char srcip[4];
    unsigned char dstmac[6];
    unsigned char dstip[4];
};

struct ip_datagram
{
    unsigned char ver_ihl;
    unsigned char tos;
    unsigned short len;
    unsigned short id;
    unsigned short flag_offs;
    unsigned char ttl;
    unsigned char proto;
    unsigned short  checksum;
    unsigned int src;
    unsigned int dst;
    unsigned char payload[1480];
};

int forge_ip(struct ip_datagram *ip, unsigned char * dst, int payloadlen,unsigned char
    proto, unsigned short fragment, int last)
{
    if(fragment>0x1FFF)
    {
        printf("[ERROR] fragment size");
        exit(1);
    }
```

```
68
69      ip->ver_ihl=0x45;
70      ip->tos=0;
71      ip->len=htons(payloadlen+20);
72      ip->id=htons(0xABCD);
73      ip->flag_offs=htons(fragment);
74
75      if(!last)
76          ip->flag_offs |= htons(0x2000);
77
78      ip->ttl=128;
79      ip->proto=proto;
80      ip->checksum=htons(0);
81      ip->src= *(unsigned int*)myip;
82      ip->dst= *(unsigned int*)dst;
83      ip->checksum =htons(checksum((unsigned char *)ip,20));
84  };
85
86  struct icmp_packet
87  {
88      unsigned char type;
89      unsigned char code;
90      unsigned short checksum;
91      unsigned short id;
92      unsigned short seq;
93      unsigned char payload[1400];
94  };
95
96  int forge_icmp(struct icmp_packet * icmp, int payloadsize)
97  {
98      int i;
99      icmp->type=8;
100     icmp->code=0;
101     icmp->checksum=htons(0);
102     icmp->id=htons(0x1234);
103     icmp->seq=htons(1);
104
105     for(i=0;i<payloadsize;i++)
106         icmp->payload[i]=i&0xFF;
107
108     icmp->checksum=htons(checksum((unsigned char*)icmp,8 + payloadsize));
109 }
110
111 int arp_resolve(unsigned char* destip, unsigned char * destmac)
112 {
113     int len,n,i;
114     unsigned char pkt[1500];
115     struct eth_frame *eth;
116     struct arp_packet *arp;
117
118     eth = (struct eth_frame *) pkt;
119     arp = (struct arp_packet *) eth->payload;
120
121     for(i=0;i<6;i++)
122         eth->dst[i]=0xff;
123
124     for(i=0;i<6;i++)
125         eth->src[i]=mymac[i];
126
127     eth->type=htons(0x0806);
128     arp->hw=htons(1);
129     arp->proto=htons(0x0800);
130     arp->hlen=6;
131     arp->plen=4;
132     arp->op=htons(1);
133
134     for(i=0;i<6;i++)
135         arp->srcmac[i]=mymac[i];
136
137     for(i=0;i<4;i++)
```

```
138         arp->srcip[i]=myip[i];
139
140     for(i=0;i<6;i++)
141         arp->dstmac[i]=0;
142
143     for(i=0;i<4;i++)
144         arp->dstip[i]=destip[i];
145
146     print_packet(pkt,14+sizeof(struct arp_packet),BOLD_CYAN);
147     sll.sll_family = AF_PACKET;
148     sll.sll_ifindex = if_nametoindex("wlp6s0");
149     len = sizeof(sll);
150     n=sendto(s,pkt,14+sizeof(struct arp_packet), 0,(struct sockaddr *)&sll,len);
151
152     if (n == -1)
153     {
154         perror("Recvfrom failed");
155         return 0;
156     }
157
158     while( 1 )
159     {
160         n=recvfrom(s,pkt,1500, 0,(struct sockaddr *)&sll,&len);
161
162         if (n == -1)
163         {
164             perror("Recvfrom failed");
165             return 0;
166         }
167
168         if (eth->type == htons (0x0806)) //it is ARP
169             if(arp->op == htons(2)) // it is a reply
170                 if(!memcmp(destip,arp->srcip,4))
171                 { // comes from our target
172                     memcpy(destmac,arp->srcmac,6);
173                     print_packet(pkt,14+sizeof(struct arp_packet),BOLD_CYAN);
174                     return 0;
175                 }
176   }
177 }
178
179 unsigned char packet[1500];
180
181 int main()
182 {
183     int i,n,len ;
184     unsigned char dstmac[6];
185
186     struct eth_frame * eth;
187     struct ip_datagram * ip;
188     struct icmp_packet * icmp;
189
190     s = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
191
192     if(s==-1)
193     {
194         perror("socket failed");
195         return 1;
196     }
197
198     /**** HOST ROUTING ****/
199     if( (*(unsigned int*)&myip) & (*(unsigned int*)&netmask) ==
200         (*(unsigned int*)&targetip) & (*(unsigned int*)&netmask))
201         arp_resolve(targetip,dstmac);
202     else
203         arp_resolve(gateway,dstmac);
204
205     printf("%sdestmac: %s", BOLD_RED, DEFAULT);
206
207     for(i=0; i<5; i++)
```

```
208             printf("%2x:",dstmac[i]);
209         printf("%2x\n", dstmac[i]);
210 
211         eth = (struct eth_frame *) packet;
212         ip = (struct ip_datagram *) eth->payload;
213         icmp = (struct icmp_packet *) ip->payload;
214 
215         for(i=0;i<6;i++)
216             eth->dst[i]=dstmac[i];
217 
218         for(i=0;i<6;i++)
219             eth->src[i]=mymac[i];
220 
221         eth->type=htons(0x0800);
222         forge_icmp(icmp, 20);
223         forge_ip(ip,targetip, 16, 1, 0, 0);
224         print_packet(packet,14+20+16,BOLD_YELLOW);
225 
226         for(i=0;i<sizeof(sll);i++)
227             ((char *)&sll)[i]=0;
228 
229         sll.sll_family=AF_PACKET;
230         sll.sll_ifindex = if_nametoindex("wlp6s0");
231         len=sizeof(sll);
232         n=sendto(s,packet,14+20+16, 0,(struct sockaddr *)&sll,len);
233 
234         if (n == -1)
235         {
236             perror("Sendto failed");
237             return 0;
238         }
239 
240         memcpy(ip->payload,(unsigned char*)icmp->payload + 8, 20-8);
241         forge_ip(ip, targetip, 20-8, 1, 2, 1);
242         print_packet(packet, 14+20+(20-8), BOLD_YELLOW);
243 
244         len = sizeof(sll);
245         n=sendto(s,packet,14+20+(20-8), 0,(struct sockaddr *)&sll,len);
246 
247         if (n == -1)
248         {
249             perror("Sendto failed");
250             return 0;
251         }
252 
253         while(1)
254         {
255             len=sizeof(sll);
256             n=recvfrom(s,packet,1500, 0,(struct sockaddr *)&sll,&len);
257 
258             if (n == -1)
259             {
260                 perror("Recvfrom failed");
261                 return 0;
262             }
263 
264             if (eth->type == htons (0x0800)) //it is IP
265                 if(ip->proto == 1) // it is ICMP
266                     if(icmp->type==0)
267                     {
268                         print_packet(packet, 14+20+8+20, BOLD_YELLOW);
269                         break;
270                     }
271         }
272 
273         return 0;
274 }
```

D.4.8 Statistics

```c
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/types.h>           /* See NOTES */
#include <sys/socket.h>
#include <linux/if_packet.h>
#include <net/ethernet.h> /* the L2 protocols */
#include <net/if.h>
#include <string.h>
#include "utility.h"

int s;
struct sockaddr_ll sll;

struct eth_frame {
unsigned char dst[6];
unsigned char src[6];
unsigned short type;
unsigned char payload[1460];
};

struct ip_datagram {
unsigned char ver_ihl;
unsigned char tos;
unsigned short len;
unsigned short id;
unsigned short flag_offs;
unsigned char ttl;
unsigned char proto;
unsigned short  checksum;
unsigned int src;
unsigned int dst;
unsigned char payload[1480];
};

unsigned char packet[1500];

int main(){
    int i,n,len, num_pkts=0;

    //Ethernet statistics
    int count_IP=0, count_ARP=0, count_3_level=0;
    //IP statistics
    int count_UDP=0, count_TCP=0, count_ICMP=0, count_other=0;

    unsigned char dstmac[6];

    struct eth_frame * eth;
    struct ip_datagram * ip;
    struct icmp_packet * icmp;

    s = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
    if(s==-1){perror("socket failed");return 1;}

    eth = (struct eth_frame *) packet;
    ip = (struct ip_datagram *) eth->payload;

    for(i=0;i<sizeof(sll);i++) ((char *)&sll)[i]=0;

    sll.sll_family=AF_PACKET;
    sll.sll_ifindex = if_nametoindex("wlp6s0");
    len=sizeof(sll);

    while(num_pkts<1000)
    {
        len=sizeof(sll);
        n=recvfrom(s,packet,1500, 0,(struct sockaddr *)&sll,&len);
        if (n == -1) {perror("Recvfrom failed"); return 0;}
        num_pkts++;
```

```
69
70          if (eth->type == htons (0x0800)) //IP datagram
71          {
72               count_IP++;
73
74               switch(ip->proto) // it is ICMP
75               {
76                    case 1: //ICMP packet
77                    {
78                         count_ICMP++;
79                         break;
80                    }
81
82                    case 6:
83                    {
84                         count_TCP++;
85                         break;
86                    }
87
88                    case 17:
89                    {
90                         count_UDP++;
91                         break;
92                    }
93
94                    default:
95                         count_other++;
96               }
97          }
98          else if(eth->type == htons(0x0806)) //ARP packet
99               count_ARP++;
100         else //Neither IP nor ARP packet
101              count_3_level++;
102     }
103
104
105     printf("%s_____________________________________________________________________%s\n",
106              BOLD_GREEN, DEFAULT);
107
108     printf("%s Ethernet statistics        IP statistics%s\n",
109              BOLD_RED, DEFAULT);
110     printf("%s   IP packets:%s %6.2lf%%        %sICMP packets:%s %6.2lf%%\n",
111              BOLD_GREEN, DEFAULT, ((double) count_IP*100.0)/1000.0, BOLD_YELLOW, DEFAULT, ((
    double) count_ICMP*100.0)/count_IP);
112     printf("%s   ARP packets:%s %6.2lf%%        %sTCP packets:%s %6.2lf%%\n",
113              BOLD_GREEN, DEFAULT, ((double) count_ARP*100.0)/1000.0, BOLD_YELLOW, DEFAULT,
    ((double) count_TCP*100.0)/count_IP);
114     printf("%sOther packets:%s %6.2lf%%        %sUDP packets:%s %6.2lf%%\n",
115              BOLD_GREEN, DEFAULT, ((double) count_3_level*100.0)/1000.0, BOLD_YELLOW,
    DEFAULT, ((double) count_UDP*100.0)/count_IP);
116     printf("                                  %sOther packets:%s %6.2lf%%\n",
117              BOLD_YELLOW, DEFAULT, ((double) count_other*100.0)/count_IP);
118
119     printf("%s_____________________________________________________________________%s\n\n",
120              BOLD_GREEN, DEFAULT);
121
122     return 0;
123 }
```

D.4.9 TCP

```
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <linux/if_packet.h>
#include <net/ethernet.h> /* the L2 protocols */
#include <net/if.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

#include "utility.h"
unsigned char myip[4]={88,80,187,84};
unsigned char netmask[4]={255,255,255,0};
unsigned char mymac[6]={0xf2,0x3c,0x91,0xdb,0xc2,0x98};
unsigned char gateway[4]={88,80,187,1};

//unsigned char targetip[4]={88,80,187,50};
unsigned char targetip[4]={147,162,2,100};
unsigned char targetmac[6];
unsigned char buffer[1500];
int s;
struct sockaddr_ll sll;

int printpacket(unsigned char *b,int l){
int i;
 for(i=0;i<l;i++){
  printf("%.2x(%.3d) ",b[i],b[i]);
  if(i%4 == 3) printf("\n");
  }
  printf("\n================\n");
}

struct eth_frame {
unsigned char dst[6];
unsigned char src[6];
unsigned short type;
unsigned char payload[1460];
};

struct arp_packet{
unsigned short hw;
unsigned short proto;
unsigned char hlen;
unsigned char plen;
unsigned short op;
unsigned char srcmac[6];
unsigned char srcip[4];
unsigned char dstmac[6];
unsigned char dstip[4];
};

struct ip_datagram {
unsigned char ver_ihl;
unsigned char tos;
unsigned short len;
unsigned short id;
unsigned short flag_offs;
unsigned char ttl;
unsigned char proto;
unsigned short checksum;
unsigned int src;
unsigned int dst;
unsigned char payload[1480];
};

int forge_ip(struct ip_datagram *ip, unsigned char * dst, int payloadlen,unsigned char
    proto)
```

```
{
ip->ver_ihl=0x45;
ip->tos=0;
ip->len=htons(payloadlen+20);
ip->id=htons(0xABCD);
ip->flag_offs=htons(0);
ip->ttl=128;
ip->proto=proto;
ip->checksum=htons(0);
ip->src= *(unsigned int*)myip;
ip->dst= *(unsigned int*)dst;
ip->checksum =htons(checksum((unsigned char *)ip,20));
/* Calculate the checksum!!!*/
};

struct tcp_segment
{
unsigned short src_port;
unsigned short dst_port;
unsigned int seq_num;
unsigned int ack_num;
unsigned short off_res_flags;
unsigned short window;
unsigned short checksum;
unsigned short urg_pointer;
unsigned int options;
unsigned char data[1376];
};

struct pseudo_header
{
    unsigned int src_IP;
    unsigned int dst_IP;
    unsigned short protocol;
    unsigned short length;
    unsigned char tcp_header[20];
};

int arp_resolve(unsigned char* destip, unsigned char * destmac)
{
int len,n,i;
unsigned char pkt[1500];
struct eth_frame *eth;
struct arp_packet *arp;

eth = (struct eth_frame *) pkt;
arp = (struct arp_packet *) eth->payload;
for(i=0;i<6;i++) eth->dst[i]=0xff;
for(i=0;i<6;i++) eth->src[i]=mymac[i];
eth->type=htons(0x0806);
arp->hw=htons(1);
arp->proto=htons(0x0800);
arp->hlen=6;
arp->plen=4;
arp->op=htons(1);
for(i=0;i<6;i++) arp->srcmac[i]=mymac[i];
for(i=0;i<4;i++) arp->srcip[i]=myip[i];
for(i=0;i<6;i++) arp->dstmac[i]=0;
for(i=0;i<4;i++) arp->dstip[i]=destip[i];
//printpacket(pkt,14+sizeof(struct arp_packet));
sll.sll_family = AF_PACKET;
sll.sll_ifindex = if_nametoindex("eth0");
len = sizeof(sll);
n=sendto(s,pkt,14+sizeof(struct arp_packet), 0,(struct sockaddr *)&sll,len);
if (n == -1) {perror("Recvfrom failed"); return 0;}
while( 1 ){
  n=recvfrom(s,pkt,1500, 0,(struct sockaddr *)&sll,&len);
  if (n == -1) {perror("Recvfrom failed"); return 0;}
  if (eth->type == htons (0x0806)) //it is ARP
    if(arp->op == htons(2)) // it is a reply
```

```
138         if(!memcmp(destip,arp->srcip,4)){ // comes from our target
139           memcpy(destmac,arp->srcmac,6);
140                   printpacket(pkt,14+sizeof(struct arp_packet));
141                   return 0;
142           }
143     }
144 }
145
146 unsigned char packet[1500];
147 struct pseudo_header pseudo_h;
148
149 int main(){
150 int i,n,len ;
151 unsigned char dstmac[6];
152
153 struct eth_frame* eth;
154 struct ip_datagram* ip;
155 struct tcp_segment* tcp;
156
157 s = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
158 if(s==-1){perror("socket failed");return 1;}
159
160 /**** HOST ROUTING ****/
161 if( (*(unsigned int*)&myip) & (*(unsigned int*)&netmask) ==
162     (*(unsigned int*)&targetip) & (*(unsigned int*)&netmask))
163   arp_resolve(targetip,dstmac);
164 else
165   arp_resolve(gateway,dstmac);
166
167 /********/
168
169 printf("destmac: ");printpacket(dstmac,6);
170
171 eth = (struct eth_frame *) packet;
172 ip = (struct ip_datagram *) eth->payload;
173 tcp = (struct tcp_segment *) ip->payload;
174
175 srand((unsigned int) time(0));
176 unsigned short port = (unsigned short) ((rand() %6000)+6000);
177
178 for(i=0;i<6;i++) eth->dst[i]=dstmac[i];
179 for(i=0;i<6;i++) eth->src[i]=mymac[i];
180 eth->type=htons(0x0800);
181 tcp->src_port=htons(8080);
182 tcp->dst_port=htons(80);
183 tcp->seq_num=htonl(10);
184 tcp->off_res_flags = htons(0x5002);
185 tcp->window = 0xffff;
186 tcp->checksum = 0;
187 tcp->urg_pointer = 0;
188 forge_ip(ip,targetip, 20, 6);
189 pseudo_h.src_IP = ip->src;
190 pseudo_h.dst_IP = ip->dst;
191 pseudo_h.length = htons(20);
192 pseudo_h.protocol = htons(6);
193 memcpy(pseudo_h.tcp_header, tcp, 20);
194 tcp->checksum=htons(checksum((unsigned char*) &pseudo_h, 32));
195 printpacket(packet,14+20+20);
196
197 for(i=0;i<sizeof(sll);i++) ((char *)&sll)[i]=0;
198
199 sll.sll_family=AF_PACKET;
200 sll.sll_ifindex = if_nametoindex("eth0");
201 len=sizeof(sll);
202 n=sendto(s,packet,14+20+20, 0,(struct sockaddr *)&sll,len);
203 if (n == -1) {perror("Recvfrom failed"); return 0;}
204
205 while( 1 ){
206   len=sizeof(sll);
207   n=recvfrom(s,packet,1500, 0,(struct sockaddr *)&sll,&len);
```

```
208     if (n == -1) {perror("Recvfrom failed"); return 0;}
209     if (eth->type == htons (0x0800)) //it is IP
210       {
211           if(ip->proto == 6) // it is TCP
212           {
213               if(tcp->src_port == htons(80) &&
214                  tcp->dst_port == htons(port) &&
215                  tcp->ack_num == htonl(11) &&
216                  ((tcp->off_res_flags & htons(0x003f))==htons(0x0012)))
217               {
218                    printf("TCP response\n");
219                    printpacket(packet, n);
220                    break;
221               }
222           }
223       }
224 }
225
226 return 0;
227
228 }
```

D.4.10 Time Exceeded

```
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <linux/if_packet.h>
#include <net/ethernet.h> /* the L2 protocols */
#include <net/if.h>
#include <string.h>
#include "utility.h"

unsigned char myip[4]={192,168,1,81};
unsigned char netmask[4]={255,255,255,0};
unsigned char mymac[6]={0x4c,0xbb,0x58,0x5f,0xb4,0xdc};
unsigned char gateway[4]={192,168,1,1};

//unsigned char targetip[4]={88,80,187,50};
unsigned char targetip[4]={216,58,212,196};
unsigned char targetmac[6];
unsigned char buffer[1500];
int s;
struct sockaddr_ll sll;

struct eth_frame
{
    unsigned char dst[6];
    unsigned char src[6];
    unsigned short type;
    unsigned char payload[1460];
};

struct arp_packet
{
    unsigned short hw;
    unsigned short proto;
    unsigned char hlen;
    unsigned char plen;
    unsigned short op;
    unsigned char srcmac[6];
    unsigned char srcip[4];
    unsigned char dstmac[6];
    unsigned char dstip[4];
};

struct ip_datagram
{
    unsigned char ver_ihl;
    unsigned char tos;
    unsigned short len;
    unsigned short id;
    unsigned short flag_offs;
    unsigned char ttl;
    unsigned char proto;
    unsigned short  checksum;
    unsigned int src;
    unsigned int dst;
    unsigned char payload[1480];
};

int forge_ip(struct ip_datagram *ip, unsigned char * dst, int payloadlen,unsigned char
    proto)
{
    ip->ver_ihl=0x45;
    ip->tos=0;
    ip->len=htons(payloadlen+20);
    ip->id=htons(0xABCD);
    ip->flag_offs=htons(0);
    ip->ttl=8;
    ip->proto=proto;
```

```
68      ip->checksum=htons(0);
69      ip->src= *(unsigned int*)myip;
70      ip->dst= *(unsigned int*)dst;
71      ip->checksum =htons(checksum((unsigned char *)ip,20));
72  };
73
74  struct icmp_packet
75  {
76      unsigned char type;
77      unsigned char code;
78      unsigned short checksum;
79      unsigned int unused;
80      unsigned char payload[84];
81  };
82
83  int forge_icmp(struct icmp_packet * icmp, int payloadsize)
84  {
85      int i;
86      icmp->type=8;
87      icmp->code=0;
88      icmp->checksum=htons(0);
89      icmp->unused = 0;
90
91      for(i=0;i<payloadsize;i++)
92          icmp->payload[i]=i&0xFF;
93
94      icmp->checksum=htons(checksum((unsigned char*)icmp,8 + payloadsize));
95  }
96
97  int arp_resolve(unsigned char* destip, unsigned char * destmac)
98  {
99      int len,n,i;
100     unsigned char pkt[1500];
101     struct eth_frame *eth;
102     struct arp_packet *arp;
103
104     eth = (struct eth_frame *) pkt;
105     arp = (struct arp_packet *) eth->payload;
106
107     for(i=0;i<6;i++)
108         eth->dst[i]=0xff;
109
110     for(i=0;i<6;i++)
111         eth->src[i]=mymac[i];
112
113     eth->type=htons(0x0806);
114
115     arp->hw=htons(1);
116     arp->proto=htons(0x0800);
117     arp->hlen=6;
118     arp->plen=4;
119     arp->op=htons(1);
120
121     for(i=0;i<6;i++)
122         arp->srcmac[i]=mymac[i];
123
124     for(i=0;i<4;i++)
125         arp->srcip[i]=myip[i];
126
127     for(i=0;i<6;i++)
128         arp->dstmac[i]=0;
129
130     for(i=0;i<4;i++)
131         arp->dstip[i]=destip[i];
132
133     print_packet(pkt,14+sizeof(struct arp_packet), BOLD_CYAN);
134
135     sll.sll_family = AF_PACKET;
136     sll.sll_ifindex = if_nametoindex("wlp6s0");
137     len = sizeof(sll);
```

```
138     n=sendto(s,pkt,14+sizeof(struct arp_packet), 0,(struct sockaddr *)&sll,len);
139
140     if (n == -1)
141     {
142         perror("Recvfrom failed");
143         return 0;
144     }
145
146     while( 1 )
147     {
148         n=recvfrom(s,pkt,1500, 0,(struct sockaddr *)&sll,&len);
149
150         if (n == -1)
151         {
152             perror("Recvfrom failed");
153             return 0;
154         }
155
156         if (eth->type == htons (0x0806)) //it is ARP
157             if(arp->op == htons(2)) // it is a reply
158                 if(!memcmp(destip,arp->srcip,4))
159                 { // comes from our target
160                     memcpy(destmac,arp->srcmac,6);
161                     print_packet(pkt,14+sizeof(struct arp_packet), BOLD_CYAN);
162                     return 0;
163                 }
164   }
165 }
166
167 unsigned char packet[1500];
168
169 int main()
170 {
171     int i,n,len;
172     unsigned char dstmac[6];
173
174     struct eth_frame* eth;
175     struct ip_datagram* ip;
176     struct icmp_packet* icmp;
177
178     s = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
179
180     if(s==-1)
181     {
182         perror("socket failed");
183         return 1;
184     }
185
186     /**** HOST ROUTING ****/
187     if( (*(unsigned int*)&myip) & (*(unsigned int*)&netmask) ==
188         (*(unsigned int*)&targetip) & (*(unsigned int*)&netmask))
189         arp_resolve(targetip,dstmac);
190     else
191         arp_resolve(gateway,dstmac);
192
193     printf("%sdestmac: %s", BOLD_RED, DEFAULT);
194
195     for(i=0; i<5; i++)
196         printf("%2x:",dstmac[i]);
197     printf("%2x\n", dstmac[i]);
198
199     eth = (struct eth_frame *) packet;
200     ip = (struct ip_datagram *) eth->payload;
201     icmp = (struct icmp_packet *) ip->payload;
202
203     for(i=0;i<6;i++)
204         eth->dst[i]=dstmac[i];
205
206     for(i=0;i<6;i++)
207         eth->src[i]=mymac[i];
```

```
208
209         eth->type=htons(0x0800);
210         forge_icmp(icmp, 20);
211         forge_ip(ip,targetip, 20+8, 1);
212         print_packet(packet,14+20+8+20, BOLD_YELLOW);
213
214         for(i=0;i<sizeof(sll);i++)
215             ((char *)&sll)[i]=0;
216
217         sll.sll_family=AF_PACKET;
218         sll.sll_ifindex = if_nametoindex("wlp6s0");
219         len=sizeof(sll);
220         n=sendto(s,packet,14+20+8+20, 0,(struct sockaddr *)&sll,len);
221
222         if (n == -1)
223         {
224             perror("Recvfrom failed");
225             return 0;
226         }
227
228         while( 1 )
229         {
230             len=sizeof(sll);
231             n=recvfrom(s,packet,1500, 0,(struct sockaddr *)&sll,&len);
232
233             if (n == -1)
234             {
235                 perror("Recvfrom failed");
236                 return 0;
237             }
238
239             if (eth->type == htons (0x0800)) //it is IP
240                 if(ip->proto == 1) // it is ICMP
241                     if(icmp->type==11 && icmp->code == 0)
242                     {
243                         print_packet(packet,n, BOLD_YELLOW);
244                         printf("_____________________________________\n");
245                         printf("%sRemote IP address:%s   ", BOLD_BLUE, DEFAULT);
246
247                         unsigned char* src_ip = (unsigned char*) &(ip->src);
248                         for(i=0; i<3; i++)
249                             printf("%u.", src_ip[i]);
250                         printf("%u\n", src_ip[i]);
251
252                         printf("_____________________________________\n");
253
254                         break;
255                     }
256         }
257
258         return 0;
259 }
```

D.4.11 Traceroute

```
#include "utility.h"
#include "traceroute.h"
#include "arp.h"

int verbose = MIN_VERBOSE;
double precision = 1000.0; //ms=1000 ns=1000000

int main(int argc, char** argv)
{
    int sd;
    int i;
    unsigned int x;
    FILE* fd;
    char command[60];
    char* interface;
    char line[LINE_SIZE];
    unsigned char network[4];
    unsigned char gateway[4];
    unsigned char mask[4];
    char mac_file[30];
    char c;
    struct hostent* he;
    struct in_addr addr;

    host src; //me
    host dst; //remote host
    int size_pkt = DEFAULT_SIZE;

    if(argc==1)
    {
        printf("You need to specify at least destination address, type --help for info");
        exit(1);
    }
    else if(argc>=2)
    {
        if(inet_aton(argv[1], &addr)==0) //input argument is not a valid IP address
        {
            he = gethostbyname(argv[1]);

            if(he == NULL)
                control(-1, "Get IP from hostname");
            else
            {
                for(i=0; i<4; i++)
                    dst.ip[i] = (unsigned char) (he->h_addr[i]);
            }

        }
        else
        {
            unsigned char *p = (unsigned char*) &(addr.s_addr);

            for(i=0; i<4; i++)
                dst.ip[i] = p[i];
        }

        if(argc>2)
        {
            int i=2;
            for(; i<argc; i++)
            {
                if(!strncmp(argv[i], "-s", 2))
                    size_pkt = atoi(argv[++i]);
                else if(!strncmp(argv[i], "-v", 2))
                    verbose  = MAX_VERBOSE;
            }
        }
    }
```

```
69
70      printf("\n%s---------------Remote  analysis----------------------\n%s", BOLD_RED,
        DEFAULT);
71      printf("%sDestination address = %s",BOLD_GREEN, DEFAULT);
72      for(i=0; i<3; i++)
73      {
74          printf("%u.", dst.ip[i]);
75      }
76      printf("%u\n", dst.ip[i]);
77
78
79      //Evaluation of Ethernet interface name
80      sprintf(command, "route -n | tac | head --lines=-2 ");
81      fd = popen(command, "r");
82
83      if(fd == NULL)
84          control(-1, "Open pipe");
85
86      while(fgets(line, LINE_SIZE, fd)!=NULL)
87      {
88          char* s = strtok(line, " ");
89          i=0;
90
91          if(s!=NULL)
92          {
93              if (inet_aton(s, &addr)!=0)
94              {
95                  unsigned char *p = (unsigned char*) &(addr.s_addr);
96
97                  memcpy(network, p, 4);
98                  //for(j=0; j<4; j++)
99                  //    network[j] = p[j];
100             }
101             i++;
102         }
103
104         while((s=strtok(NULL," "))!=NULL && i<8)
105         {
106             switch(i)
107             {
108                 case ROUTE_GATEWAY_INDEX:
109                 {
110                     if (inet_aton(s, &addr)!=0)
111                     {
112                         unsigned char *p = (unsigned char*) &(addr.s_addr);
113
114                         memcpy(gateway, p, 4);
115                         //for(j=0; j<4; j++)
116                         //    gateway[j] = p[j];
117                     }
118                     break;
119                 }
120
121                 case ROUTE_MASK_INDEX:
122                 {
123                     if (inet_aton(s, &addr)!=0)
124                     {
125                         unsigned char *p = (unsigned char*) &(addr.s_addr);
126
127                         memcpy(mask,p, 4);
128                         //for(j=0; j<4; j++)
129                         //  mask[j] = p[j];
130                     }
131                     break;
132                 }
133
134                 case ROUTE_INTERFACE_INDEX:
135                 {
136                     s[strlen(s)-1]=0;
137                     interface = s;
```

```
138                 }
139             }
140
141             i++;
142         }
143
144
145         if((*(unsigned int*) &network)==((*((unsigned int*) &(dst.ip))) & (*((unsigned int
    *) &mask))))
146         {
147             break;
148         }
149     }
150
151     printf("\n");
152     printf("%sGateway: %s", BOLD_MAGENTA, DEFAULT);
153     for(i=0; i<3; i++)
154         printf("%u.", gateway[i]);
155     printf("%u\n", gateway[i]);
156
157     printf("%sNetwork: %s", BOLD_MAGENTA, DEFAULT);
158     for(i=0; i<3; i++)
159         printf("%u.", network[i]);
160     printf("%u\n", network[i]);
161
162     printf("%s   Mask: %s", BOLD_MAGENTA, DEFAULT);
163     for(i=0; i<3; i++)
164         printf("%u.", mask[i]);
165     printf("%u\n", mask[i]);
166
167
168     //See the MAC address of eth0 looking to e.g. "/sys/class/net/eth0/address" content
169     sprintf(mac_file, MAC_DEFAULT_FILE, interface);
170
171     fd = fopen(mac_file, "r");
172
173     for(i=0; i<5; i++)
174     {
175         fscanf(fd, "%x:", &x);
176         src.mac[i]=(unsigned char) x;
177     }
178
179     fscanf(fd, "%x\n", &x);
180     src.mac[i]=(unsigned char) x;
181
182     fclose(fd);
183
184     printf("\n");
185
186     printf("%sEthernet Interface:%s %s\n", BOLD_CYAN, DEFAULT, interface);
187
188     printf("%sSource MAC address: %s", BOLD_CYAN, DEFAULT);
189     for(i=0; i<5; i++)
190         printf("%x:", src.mac[i]);
191     printf("%x\n", src.mac[i]);
192
193
194     //Evaluation of IPv4 address of ethernet interface in input
195     sprintf(command, "ip -4 addr show %s | grep -oP '(?<=inet\\s)\\d+(\\.\\d+){3}'",
    interface);
196     fd = popen(command, "r");
197
198     for(i=0; i<3; i++)
199     {
200         fscanf(fd, "%u%c", &x, &c);
201         src.ip[i]=x;
202     }
203
204     fscanf(fd, "%u", &x);
205     src.ip[i]=x;
```

```
206
207     pclose(fd);
208
209     printf("%sSource IP address: %s", BOLD_CYAN, DEFAULT);
210     for(i=0; i<3; i++)
211         printf("%d.", src.ip[i]);
212     printf("%d\n",src.ip[i]);
213
214
215     //Creation of the socket
216     sd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
217     control(sd, "Socket failed\n");
218
219     //ARP resolution
220     /*
221         if(myip & mask == dstip & mask)
222             arp_resolution(sd, &dst, 0.0.0.0);
223         else
224             arp_resolution(sd, &dst, gateway);
225     */
226     printf("\n%s------------------ARP packets----------------------\n%s", BOLD_RED,
    DEFAULT);
227     arp_resolution(sd, &src, &dst, interface, gateway, verbose);
228
229     printf("%sDestination MAC address: %s", BOLD_YELLOW, DEFAULT);
230     for(i=0; i<5; i++)
231         printf("%x:", dst.mac[i]);
232     printf("%x\n", dst.mac[i]);
233
234     //Traceroute application
235     printf("\n%s--------------------------------Traceroute
    ---------------------------------\n%s", BOLD_RED, DEFAULT);
236     traceroute(sd, size_pkt, interface, src, dst);
237
238     printf("%s%s%s\n", BOLD_RED, LINE_32_BITS, DEFAULT);
239     return 0;
240 }
241
242 void traceroute(int sd, int size_pkt, char* interface, host src, host dst)
243 {
244     unsigned char ttl=0;
245     int time_exceeded = 1;
246     int count_hop = 0;
247
248     while(time_exceeded)
249     {
250         time_exceeded = traceroute_iteration(sd, &count_hop, ttl+1, size_pkt, interface,
    src, dst);
251         ttl++;
252     }
253
254     printf("\n %sNUMBER OF HOPS:%s %d\n", BOLD_YELLOW, DEFAULT, count_hop);
255 }
256
257 int traceroute_iteration(int sd, int* id_pkt, unsigned char ttl, int size_pkt, char*
    interface, host src, host dst)
258 {
259     unsigned char packet[PACKET_SIZE];
260     struct sockaddr_ll sll;
261     eth_frame *eth;
262     ip_datagram *ip;
263     icmp_pkt *icmp;
264     int i;
265     socklen_t len;
266     int n;
267
268     eth = (eth_frame*) packet;
269     ip = (ip_datagram*) (eth->payload);
270     icmp = (icmp_pkt*) (ip->payload);
271
```

```
272     //Ethernet header
273     memcpy(eth->src, src.mac, 6);
274     memcpy(eth->dst, dst.mac, 6);
275     eth->type = htons(0x0800);
276
277     //IP packet
278     ip->ver_IHL = 0x45;
279     ip->type_service = 0;
280     ip->length = htons(IP_HEADER_SIZE+ECHO_HEADER_SIZE+size_pkt);
281     ip->id = htons(0xABCD);
282     ip->flag_offs = htons(0);
283     ip->ttl = ttl;
284     ip->protocol = 1; //ICMP
285     ip->checksum = htons(0);
286     memcpy((unsigned char*) &(ip->src_IP), src.ip, 4);
287     memcpy((unsigned char*) &(ip->dst_IP), dst.ip, 4);
288     ip->checksum = htons(checksum((unsigned char*) ip, IP_HEADER_SIZE)); //Checksum of ip
        header
289
290
291     //Echo request (ICMP)
292     icmp->type = 8; //ECHO request
293     icmp->code = 0;
294     icmp->checksum = htons(0);
295     icmp->id = htons(0x1234);
296     icmp->seq = htons(ttl);
297
298     for(i=0; i<size_pkt; i++)
299         icmp->payload[i] = i & 0xff;
300
301     //Checksum of the entire packet
302     icmp->checksum = htons(checksum((unsigned char*) icmp, ECHO_HEADER_SIZE+size_pkt));
303
304     for(i=0; i<sizeof(sll);i++)
305         ((char*) &sll)[i]=0;
306
307     sll.sll_family = AF_PACKET;
308     sll.sll_ifindex = if_nametoindex(interface);
309     len = sizeof(sll);
310
311     if(verbose>50)
312     {
313         printf("\n%s                    ECHO request\n%s", BOLD_BLUE, DEFAULT);
314         print_packet(packet, ETH_HEADER_SIZE+IP_HEADER_SIZE+ECHO_HEADER_SIZE+size_pkt,
        BOLD_BLUE);
315     }
316
317     n = sendto(sd, packet, ETH_HEADER_SIZE+IP_HEADER_SIZE+ECHO_HEADER_SIZE+size_pkt, 0, (
        struct sockaddr*) &sll, len);
318     control(n,"ECHO sendto ERROR");
319
320     time_t start = clock();
321
322     len = sizeof(sll);
323     //printf("Receiving\n");
324     n = recvfrom(sd, packet, PACKET_SIZE, 0, (struct sockaddr*) &sll, &len);
325     control(n, "ECHO recvfrom ERROR");
326
327     time_t end = clock();
328     double elapsed_time = ((double) (end-start)/(double) CLOCKS_PER_SEC)*precision;
329
330     if(eth->type == htons(0x0800) && //IP datagram
331        ip->protocol == 1 && //ICMP packet
332        icmp->code == 0 &&
333        icmp->type == 0) //ECHO reply
334     {
335         if(verbose>50)
336         {
337             printf("\n%s                    ECHO reply\n%s", BOLD_BLUE, DEFAULT);
338             print_packet(packet, ETH_HEADER_SIZE+IP_HEADER_SIZE+ECHO_HEADER_SIZE+
```

```
    IP_HEADER_SIZE, BOLD_BLUE);
339         }
340
341         host hop;
342         memcpy(hop.ip, (unsigned char*) &(ip->src_IP), 4);
343         memcpy(hop.mac, eth->src, 6);
344
345         (*id_pkt)++;
346         print_route(*id_pkt, hop, elapsed_time);
347         return 0;
348     }
349     else if(eth->type == htons(0x0800) && //IP datagram
350        ip->protocol == 1 && //ICMP packet
351        icmp->type == 11 &&
352        icmp->code == 0) //ICMP Time Exceeded
353     {
354         if(verbose>50)
355         {
356             printf("\n%s                    Time Exceeded\n%s", BOLD_BLUE, DEFAULT);
357             print_packet(packet, ETH_HEADER_SIZE+IP_HEADER_SIZE+ECHO_HEADER_SIZE, BOLD_BLUE
    );
358         }
359
360         host hop;
361         memcpy(hop.ip, (unsigned char*) &(ip->src_IP), 4);
362         memcpy(hop.mac, eth->src, 6);
363         (*id_pkt)++;
364         print_route(*id_pkt, hop, elapsed_time);
365     }
366
367     return 1;
368 }
369
370 void print_route(int id, host hop, double elapsed_time)
371 {
372     int i;
373     struct hostent *host_info;
374     struct in_addr addr;
375
376     printf("%s[Hop %3d]%s at %s", BOLD_CYAN, id, DEFAULT, MAGENTA);
377
378     for(i=0; i<3; i++)
379         printf("%u.", hop.ip[i]);
380     printf("%u ", hop.ip[i]);
381
382     printf("%s", GREEN);
383
384     //Host name
385     addr.s_addr = *(uint32_t*) &(hop.ip);
386     host_info = gethostbyaddr((const void*) &addr, sizeof(addr), AF_INET);
387
388     if(host_info ==NULL)
389         printf("%s (*)      ", DEFAULT);
390     else
391         printf("%s (%s)      ", DEFAULT, host_info->h_name);
392
393     printf("%selapsed_time:%s %.3lf ", YELLOW, DEFAULT, elapsed_time);
394
395     if(precision==1.0)
396         printf("%s\n",TIME_s);
397     else if(precision==1000.0)
398         printf("%s\n",TIME_ms);
399     else if(precision==1000000.0)
400         printf("%s\n",TIME_ns);
401 }
```

D.4.12 Unreachable Destination

```
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
#include <linux/if_packet.h>
#include <net/ethernet.h> /* the L2 protocols */
#include <net/if.h>
#include <string.h>
#include "utility.h"

unsigned char myip[4]={88,80,187,84};
unsigned char netmask[4]={255,255,255,0};
unsigned char mymac[6]={0xf2,0x3c,0x91,0xdb,0xc2,0x98};
unsigned char gateway[4]={88,80,187,1};

//unsigned char targetip[4]={88,80,187,50};
unsigned char targetip[4]={10,20,30,40};
unsigned char targetmac[6];
unsigned char buffer[1500];
int s;
struct sockaddr_ll sll;

struct eth_frame
{
unsigned char dst[6];
unsigned char src[6];
unsigned short type;
unsigned char payload[1460];
};

struct arp_packet
{
unsigned short hw;
unsigned short proto;
unsigned char hlen;
unsigned char plen;
unsigned short op;
unsigned char srcmac[6];
unsigned char srcip[4];
unsigned char dstmac[6];
unsigned char dstip[4];
};

struct ip_datagram
{
unsigned char ver_ihl;
unsigned char tos;
unsigned short len;
unsigned short id;
unsigned short flag_offs;
unsigned char ttl;
unsigned char proto;
unsigned short  checksum;
unsigned int src;
unsigned int dst;
unsigned char payload[1480];
};

int forge_ip(struct ip_datagram *ip, unsigned char * dst, int payloadlen,unsigned char
    proto)
{
ip->ver_ihl=0x45;
ip->tos=0;
ip->len=htons(payloadlen+20);
ip->id=htons(0xABCD);
ip->flag_offs=htons(0);
ip->ttl=128;
ip->proto=proto;
```

```
ip->checksum=htons(0);
ip->src= *(unsigned int*)myip;
ip->dst= *(unsigned int*)dst;
ip->checksum =htons(checksum((unsigned char *)ip,20));
/* Calculate the checksum!!!*/
};

struct icmp_packet
{
unsigned char type;
unsigned char code;
unsigned short checksum;
unsigned int unused;
unsigned char payload[84];
};

int forge_icmp(struct icmp_packet * icmp, int payloadsize)
{
int i;
icmp->type=8;
icmp->code=0;
icmp->checksum=htons(0);
icmp->unused = 0;
for(i=0;i<payloadsize;i++)icmp->payload[i]=i&0xFF;
icmp->checksum=htons(checksum((unsigned char*)icmp,8 + payloadsize));
}

int arp_resolve(unsigned char* destip, unsigned char * destmac)
{
int len,n,i;
unsigned char pkt[1500];
struct eth_frame *eth;
struct arp_packet *arp;

eth = (struct eth_frame *) pkt;
arp = (struct arp_packet *) eth->payload;
for(i=0;i<6;i++) eth->dst[i]=0xff;
for(i=0;i<6;i++) eth->src[i]=mymac[i];
eth->type=htons(0x0806);
arp->hw=htons(1);
arp->proto=htons(0x0800);
arp->hlen=6;
arp->plen=4;
arp->op=htons(1);
for(i=0;i<6;i++) arp->srcmac[i]=mymac[i];
for(i=0;i<4;i++) arp->srcip[i]=myip[i];
for(i=0;i<6;i++) arp->dstmac[i]=0;
for(i=0;i<4;i++) arp->dstip[i]=destip[i];
print_packet(pkt,14+sizeof(struct arp_packet), BOLD_CYAN);
sll.sll_family = AF_PACKET;
sll.sll_ifindex = if_nametoindex("eth0");
len = sizeof(sll);
n=sendto(s,pkt,14+sizeof(struct arp_packet), 0,(struct sockaddr *)&sll,len);
if (n == -1) {perror("Recvfrom failed"); return 0;}
while( 1 ){
  n=recvfrom(s,pkt,1500, 0,(struct sockaddr *)&sll,&len);
  if (n == -1) {perror("Recvfrom failed"); return 0;}
  if (eth->type == htons (0x0806)) //it is ARP
    if(arp->op == htons(2)) // it is a reply
      if(!memcmp(destip,arp->srcip,4)){ // comes from our target
        memcpy(destmac,arp->srcmac,6);
                print_packet(pkt, 14+sizeof(struct arp_packet), BOLD_CYAN);
                return 0;
        }
  }
}

unsigned char packet[1500];

int main(){
```

```
138 int i,n,len ;
139 unsigned char dstmac[6];
140
141 struct eth_frame * eth;
142 struct ip_datagram * ip;
143 struct icmp_packet * icmp;
144
145 s = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
146 if(s==-1){perror("socket failed");return 1;}
147
148 /**** HOST ROUTING ****/
149 if( (*(unsigned int*)&myip) & (*(unsigned int*)&netmask) ==
150     (*(unsigned int*)&targetip) & (*(unsigned int*)&netmask))
151   arp_resolve(targetip,dstmac);
152 else
153   arp_resolve(gateway,dstmac);
154
155     printf("%sdestmac: %s", BOLD_RED, DEFAULT);
156
157     for(i=0; i<5; i++)
158         printf("%2x:",dstmac[i]);
159     printf("%2x\n", dstmac[i]);
160
161     eth = (struct eth_frame *) packet;
162     ip = (struct ip_datagram *) eth->payload;
163     icmp = (struct icmp_packet *) ip->payload;
164
165     for(i=0;i<6;i++) eth->dst[i]=dstmac[i];
166     for(i=0;i<6;i++) eth->src[i]=mymac[i];
167     eth->type=htons(0x0800);
168     forge_icmp(icmp, 20);
169     forge_ip(ip,targetip, 20+8, 1);
170     print_packet(packet,14+20+8+20,BOLD_YELLOW);
171
172     for(i=0;i<sizeof(sll);i++) ((char *)&sll)[i]=0;
173
174     sll.sll_family=AF_PACKET;
175     sll.sll_ifindex = if_nametoindex("eth0");
176     len=sizeof(sll);
177     n=sendto(s,packet,14+20+8+20, 0,(struct sockaddr *)&sll,len);
178     if (n == -1) {perror("Recvfrom failed"); return 0;}
179
180     while( 1 ){
181         len=sizeof(sll);
182         n=recvfrom(s,packet,1500, 0,(struct sockaddr *)&sll,&len);
183         if (n == -1) {perror("Recvfrom failed"); return 0;}
184         if (eth->type == htons (0x0800)) //it is IP
185             if(ip->proto == 1) // it is ICMP
186                 if(icmp->type==3){
187                     print_packet(packet,n, BOLD_YELLOW);
188                     printf("%s_____________________________________________________________\n%s", BOLD_BLUE,
    BOLD_RED);
189
190                     unsigned char* src_ip = (unsigned char*) &(ip->src);
191
192                     for(i=0; i<3; i++)
193                         printf("%u.", src_ip[i]);
194                     printf("%u]%s The host with address %s", src_ip[i], DEFAULT,
    BOLD_YELLOW);
195                     for(i=0; i<3; i++)
196                         printf("%u.", targetip[i]);
197                     printf("%u%s is unreachable\n", targetip[i], DEFAULT);
198
199                     printf("%s_____________________________________________________________\n%s", BOLD_BLUE,
    DEFAULT);
200
201                     break;
202                     }
```

```
203     }
204
205     return 0;
206 }
```

D.4.13 Colors

```
1 #define BOLD_RED "\033[1;31m"
2 #define BOLD_GREEN "\033[1;32m"
3 #define BOLD_YELLOW "\033[1;33m"
4 #define BOLD_BLUE "\033[1;34m"
5 #define BOLD_MAGENTA "\033[1;35m"
6 #define BOLD_CYAN "\033[1;36m"
7 #define DEFAULT "\033[0m"
```

References

[1] Arp. https://tools.ietf.org/html/rfc826.

[2] Dns. https://tools.ietf.org/html/rfc1034.

[3] Ethernet types. https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml.

[4] Http/1.0. https://tools.ietf.org/html/rfc1945.

[5] Http/1.1. https://tools.ietf.org/html/rfc2616.

[6] Https. https://tools.ietf.org/html/rfc2817.

[7] Icmp. https://tools.ietf.org/html/rfc792.

[8] Internet protocol. https://tools.ietf.org/html/rfc791.

[9] Ip upper layer protcols. https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml.

[10] Tcp. https://tools.ietf.org/html/rfc793.

[11] Udp. https://tools.ietf.org/html/rfc768.

[12] Uri. https://tools.ietf.org/html/rfc3986.

[13] Uri schemes. https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml.