PADUA UNIVERSITY

ENGINEERING COURSE

Master of Computer Engineering

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.

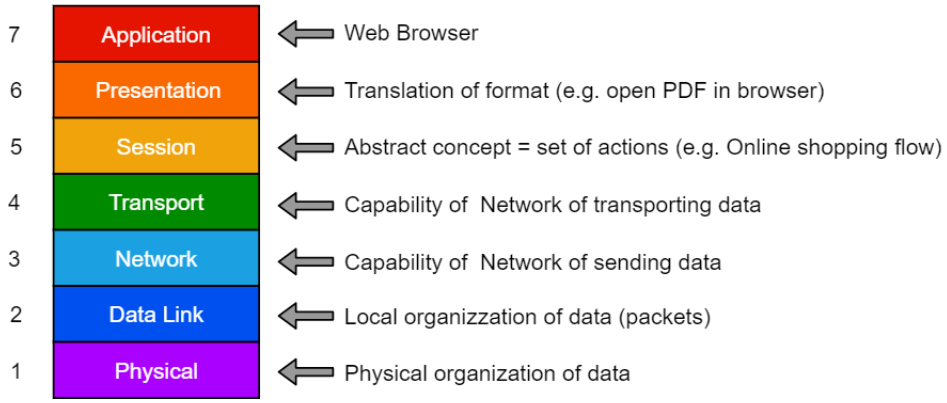
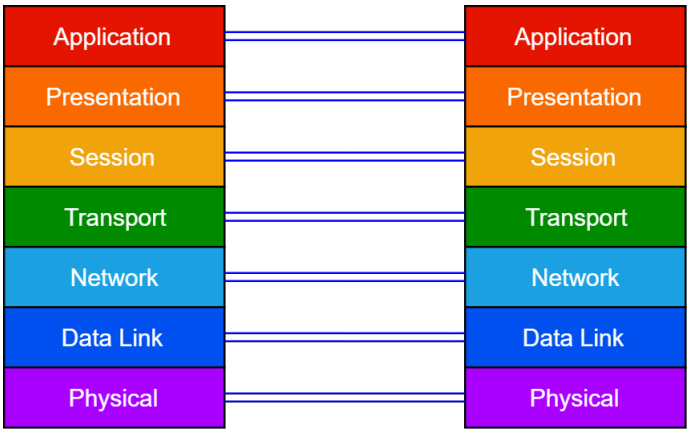


Figure 1.1: OSI model.

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

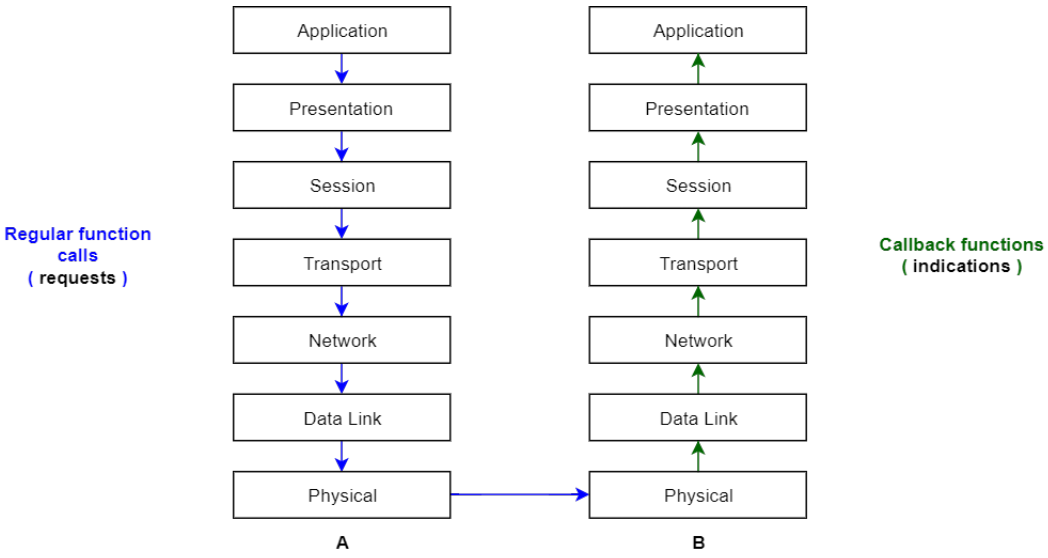


Figure 1.2: Request from A to B.

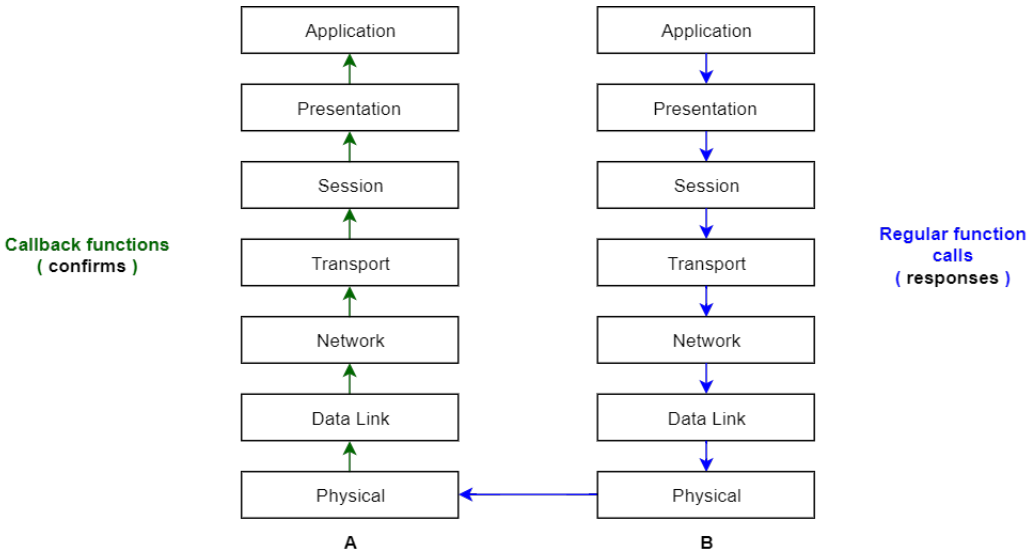


Figure 1.3: Response from B to A.

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.

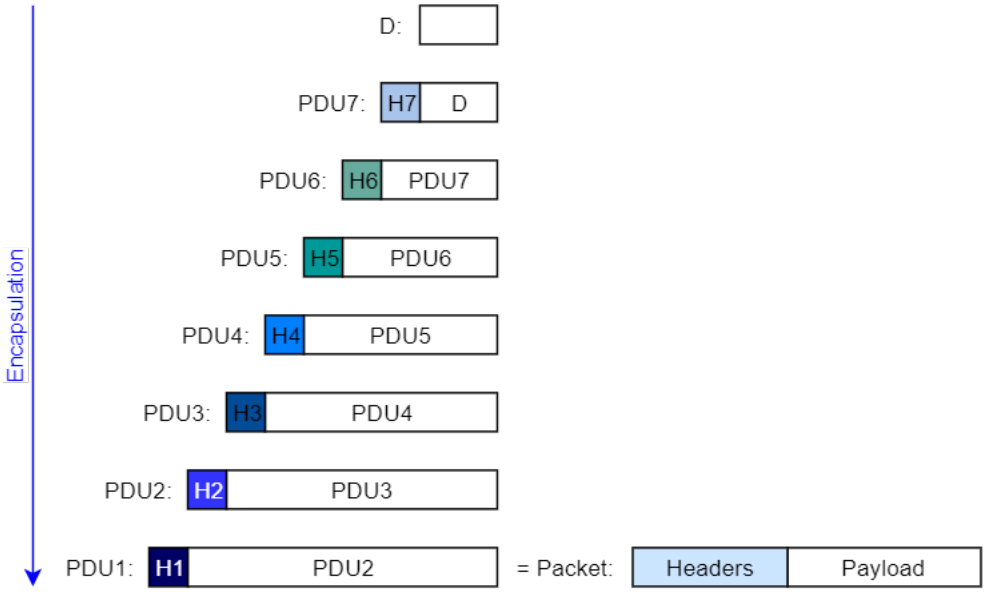


Figure 1.5: Encapsulation.

1.4 Onion model

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

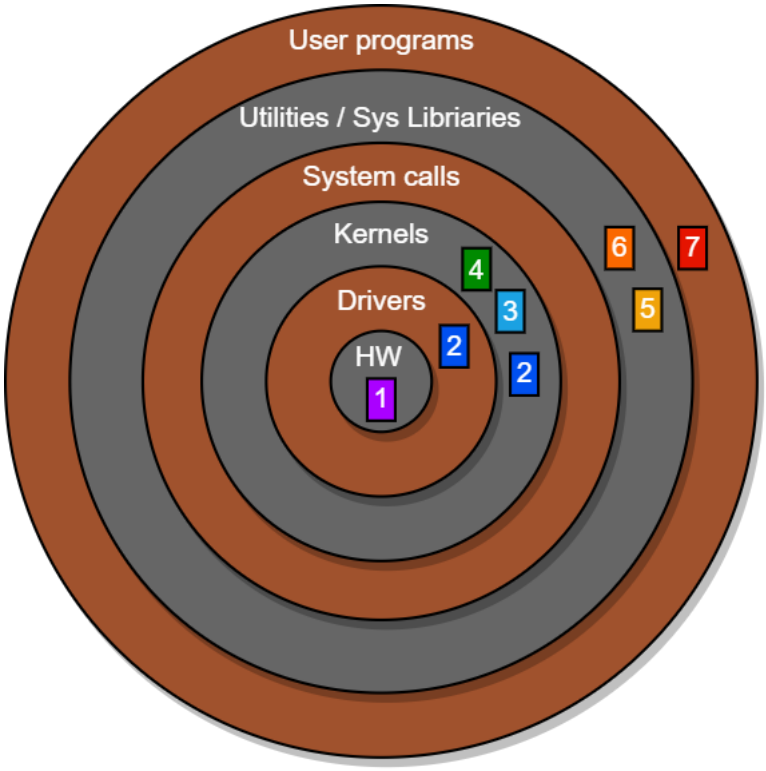
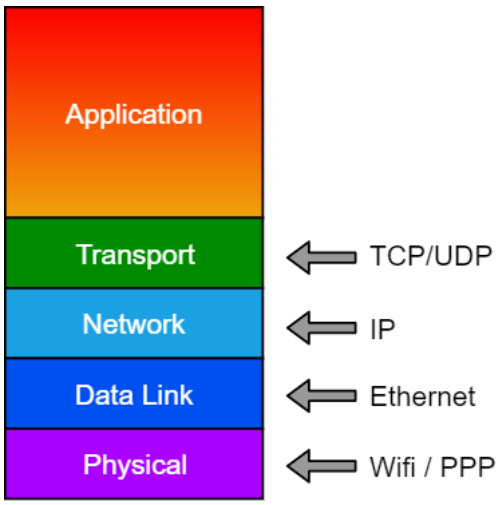


Figure 1.6: Onion model.

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)

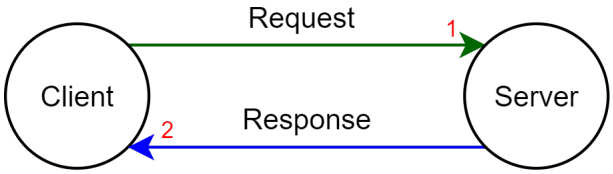


Figure 1.7: Client-Server architecture.

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.

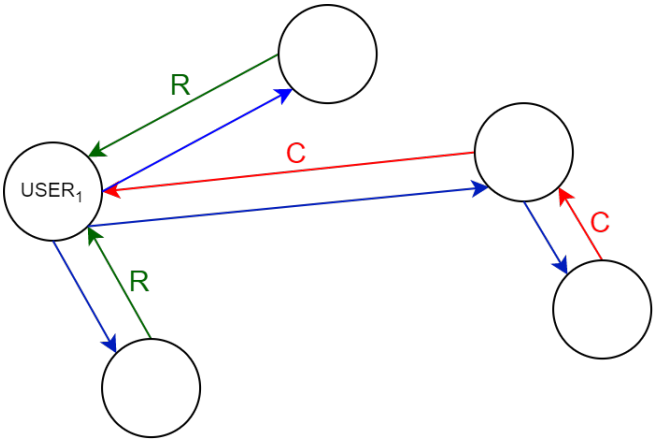


Figure 1.8: P2P architecture.

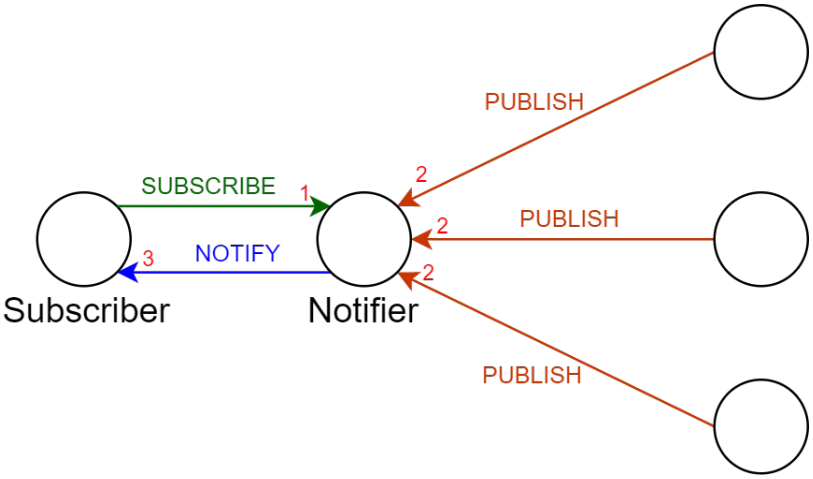


Figure 1.9: Publish/Subscribe/Notify architecture.

1.7 Types of packets

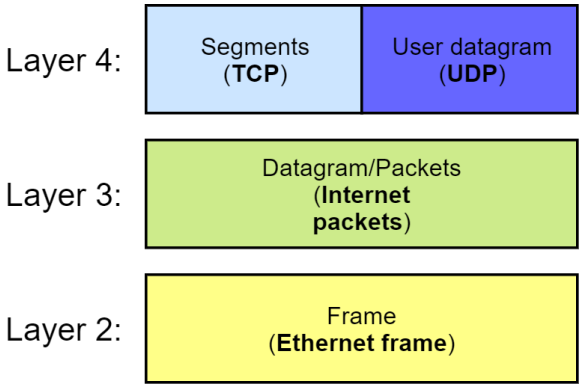


Figure 1.10: Standard names of packets.

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.

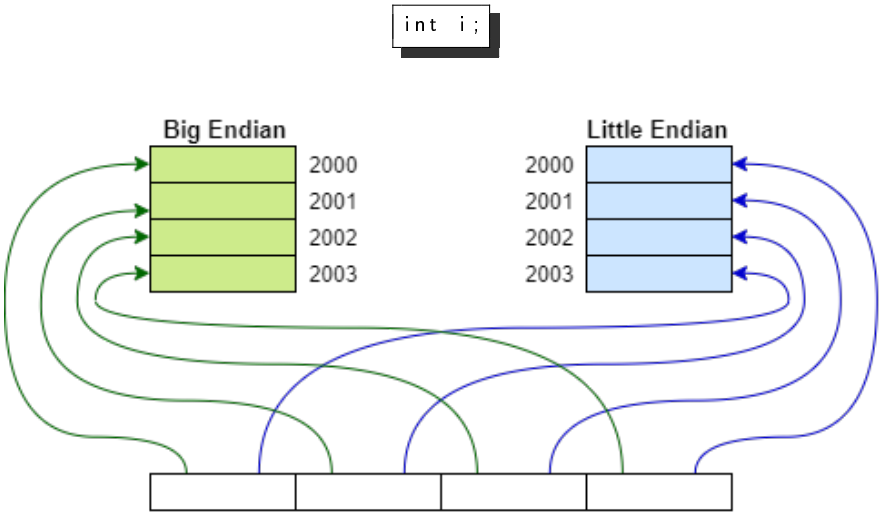


Figure 2.1: Little Endian and Big Endian.

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).

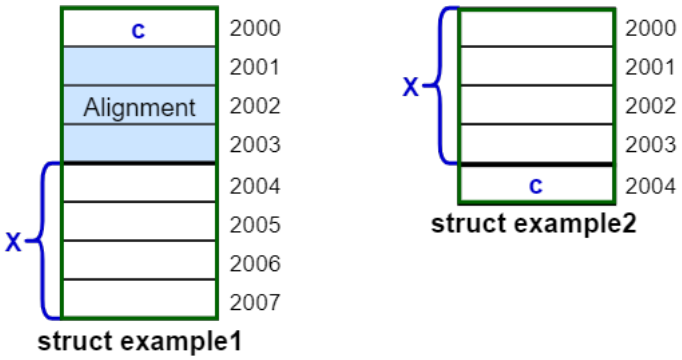
signed	unsigned
int8_t	uint8_t
int16_t	uint16_t
int32_t	uint32_t
int64_t	uint64_t

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;
}                        }
```



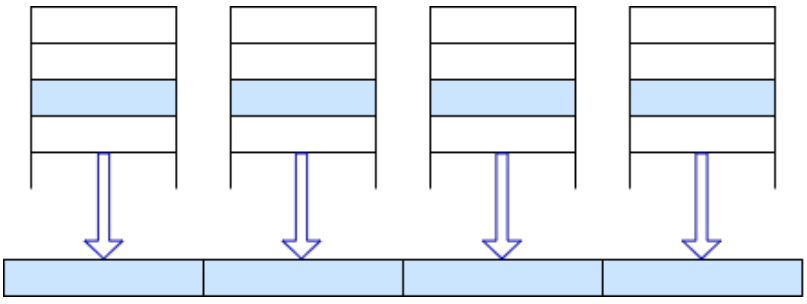


Figure 2.2: Parallel reading in one istruction in 32 bits architecture.

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

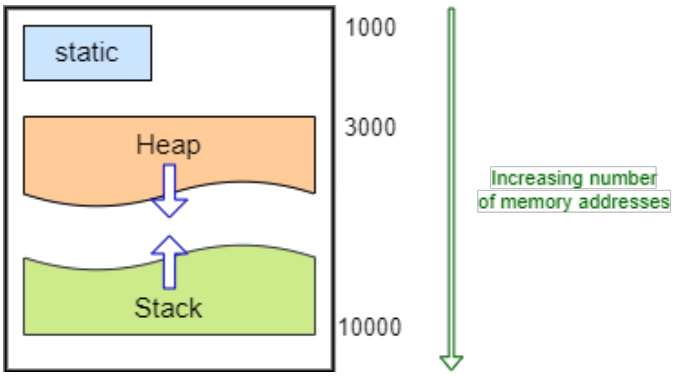


Figure 2.3: Structure of the program.

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.

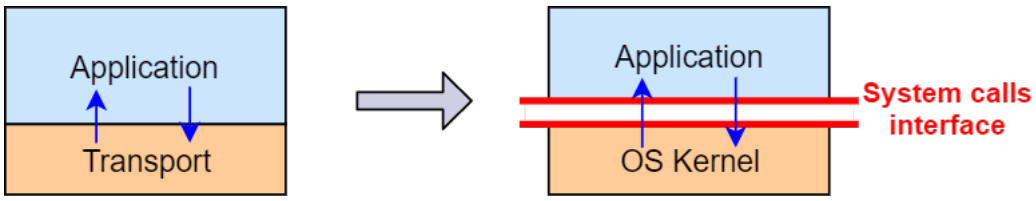


Figure 3.1: System calls interface.

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

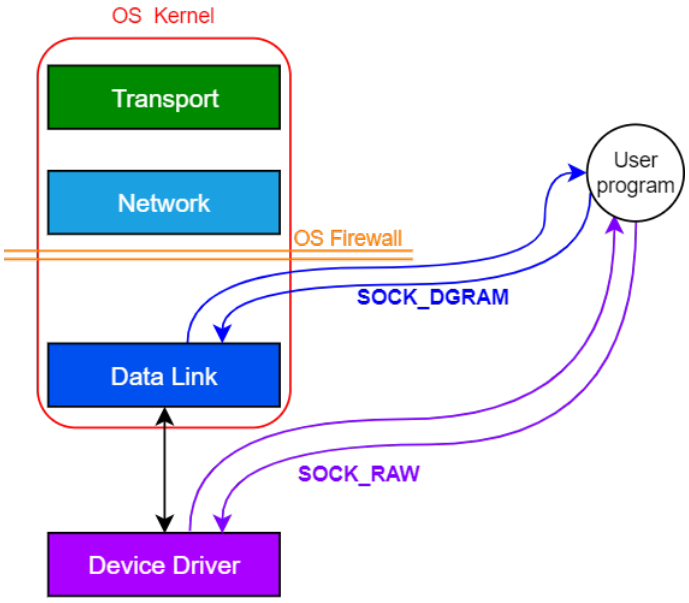


Figure 3.2: UNIX management.

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 10.2 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);
```

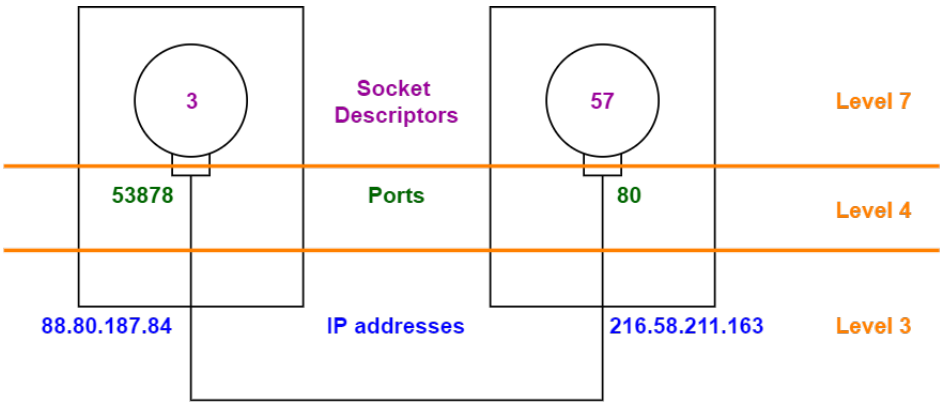


Figure 3.3: After successful connection.

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.

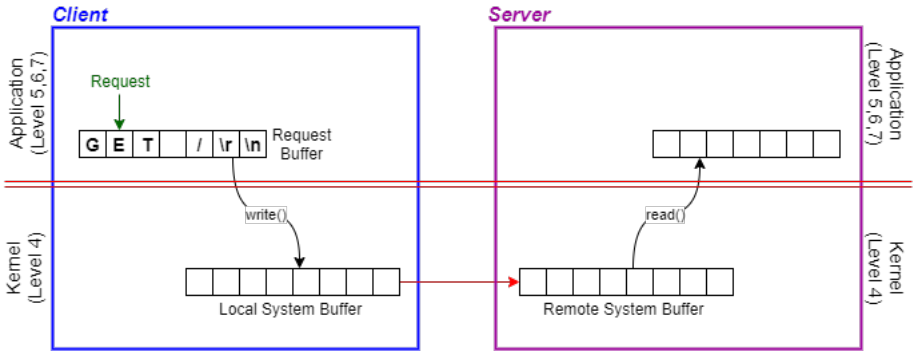


Figure 3.4: Request by the client.

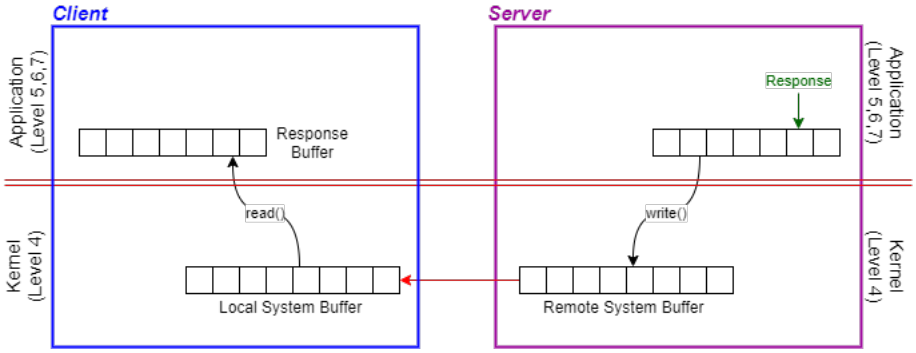


Figure 3.5: Response from the server.

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.

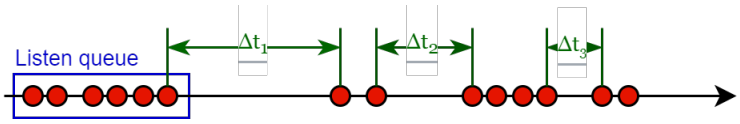


Figure 3.6: Poisson distribution of connections by clients.

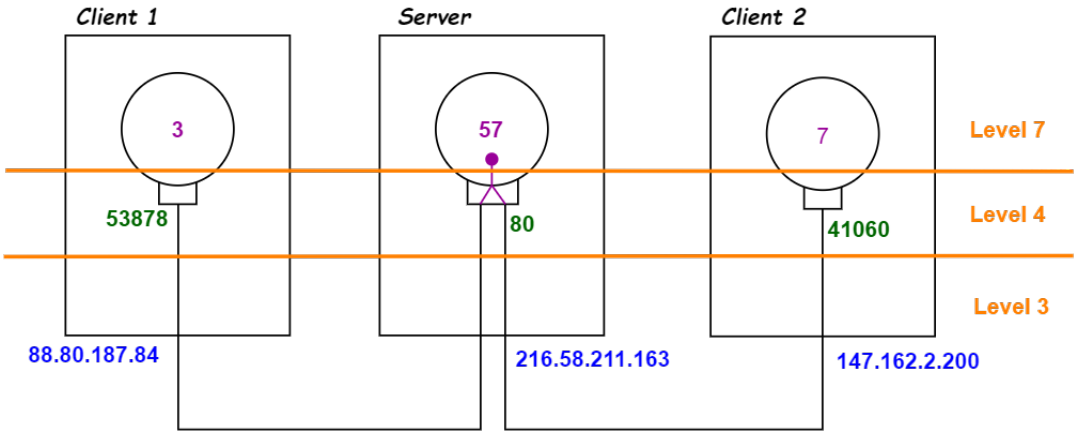


Figure 3.7: listen() function.

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).

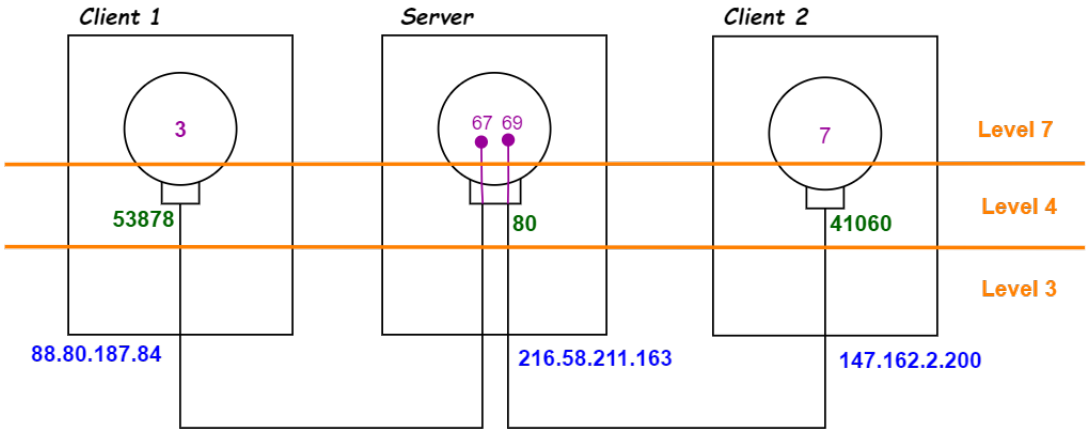


Figure 3.8: accept() function.

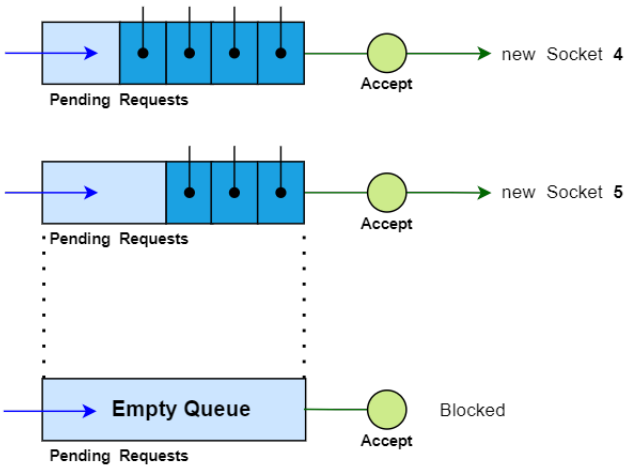


Figure 3.9: Management of pending requests with accept().

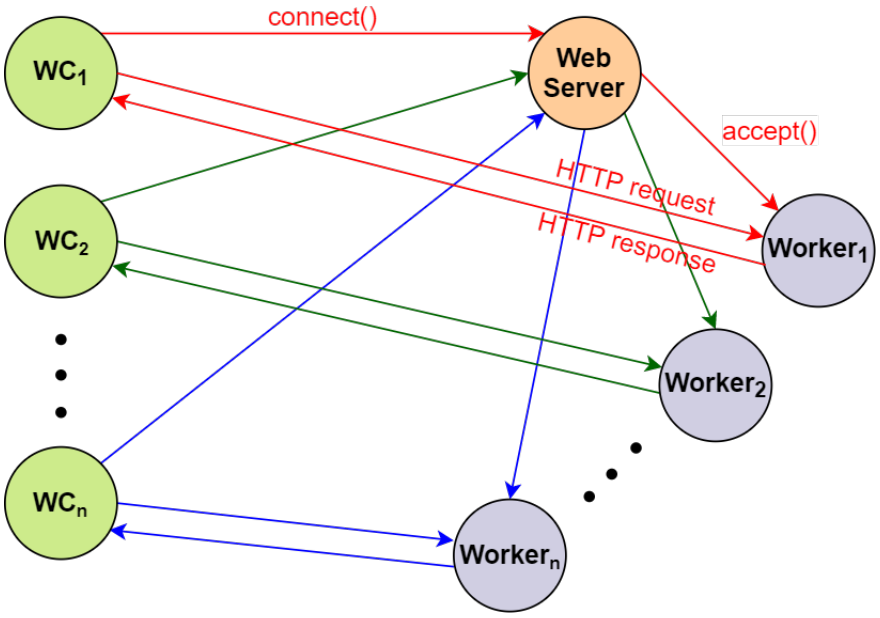


Figure 3.10: connect() and accept() functions in parallel server implementation.

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.

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).

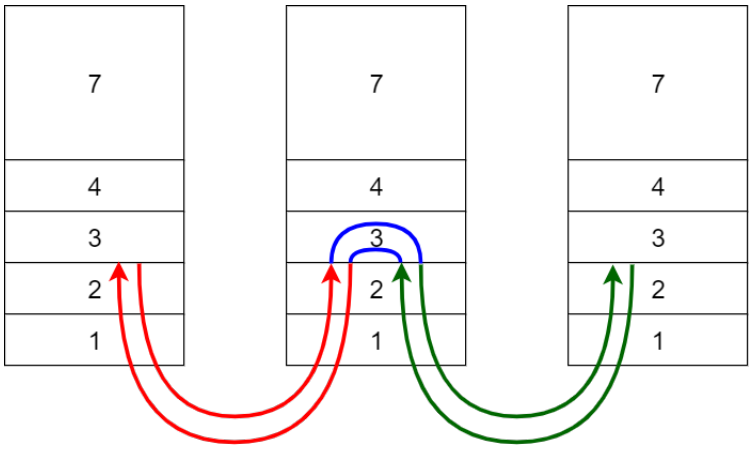


Figure 4.1: Router (Layer-3 gateway).

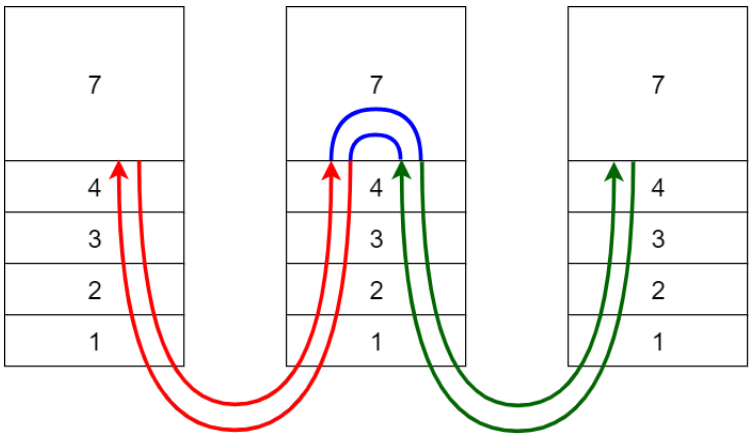


Figure 4.2: Proxy (Layer-7 gateway).

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:

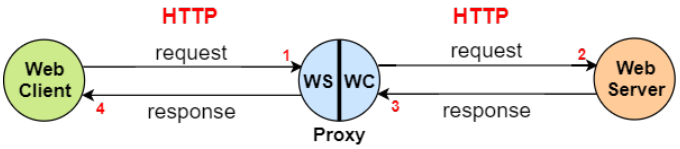


Figure 4.3: Example of proxy use.

- **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.

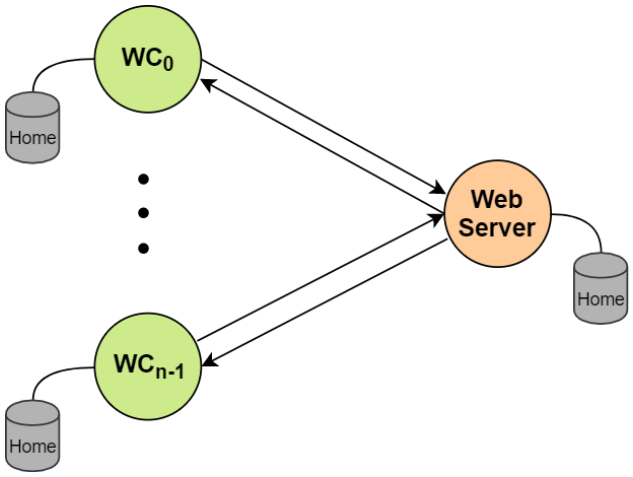


Figure 4.4: Example of caching without proxy.

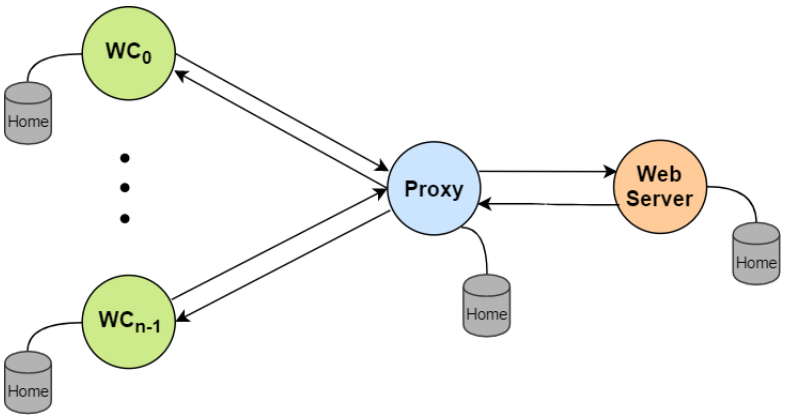


Figure 4.5: Example of caching using proxy.

- **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.

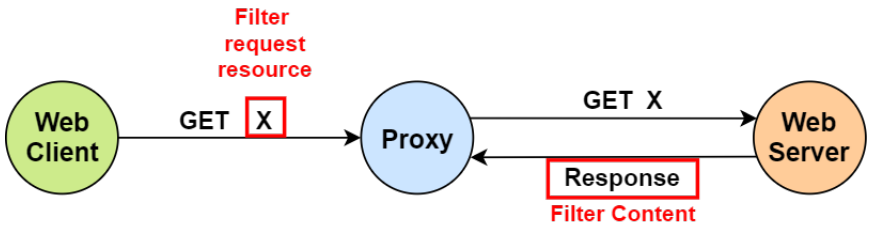


Figure 4.6: Example of proxy filtering.

- **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).

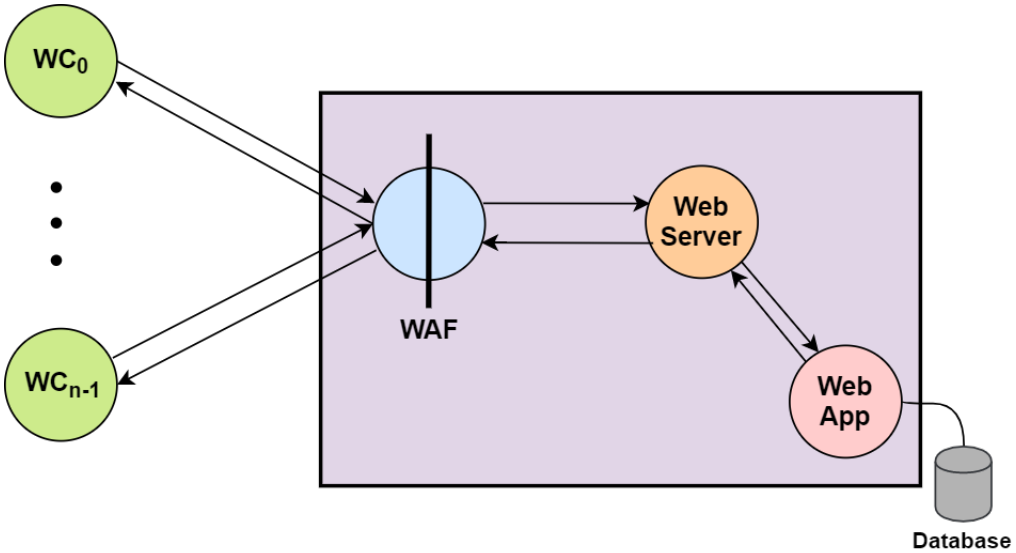


Figure 4.7: Example of WAF use.

- **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.

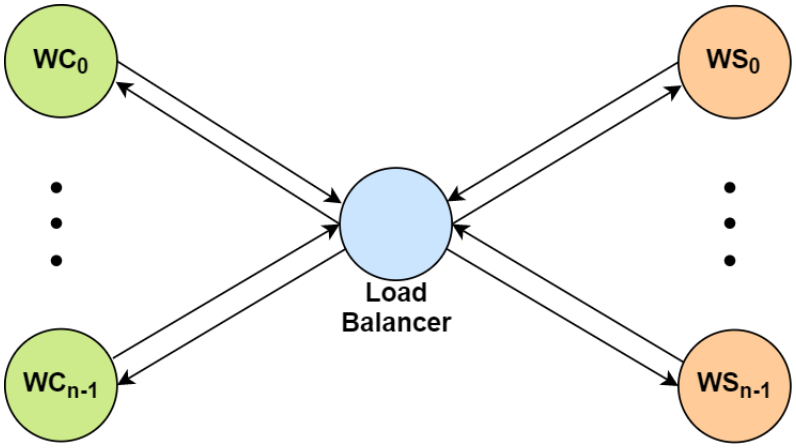


Figure 4.8: Example of load balancing through proxy.

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 Local*).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.

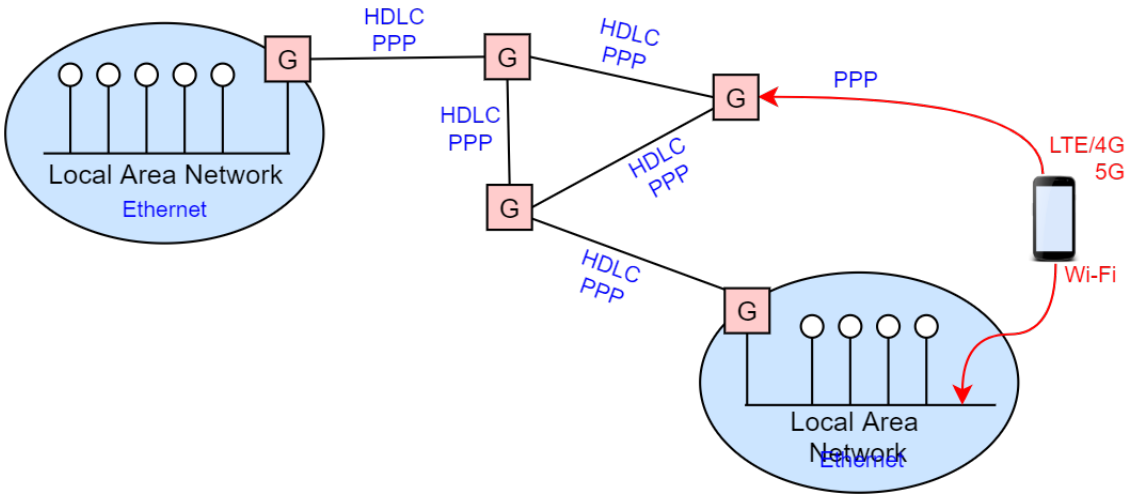


Figure 5.1: Nowadays L2 connections.

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.

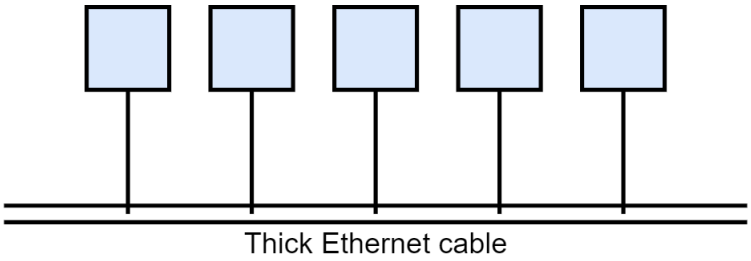


Figure 5.2: Ethernet.

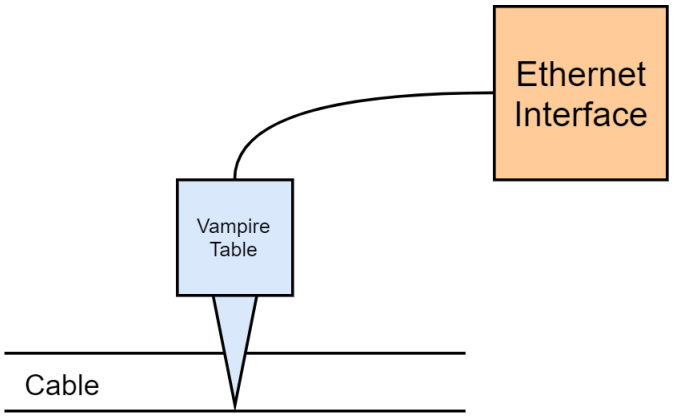


Figure 5.3: Vampire tap.

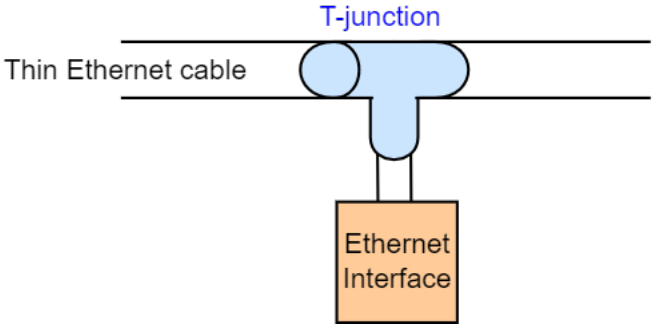


Figure 5.4: T-junction.

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

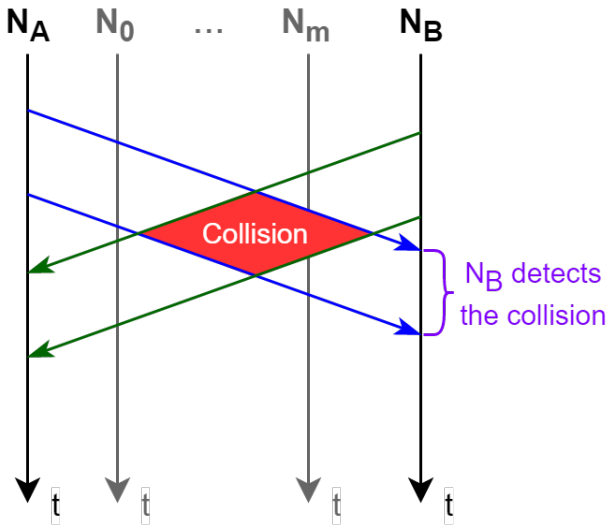


Figure 5.5: Collision detection.

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.

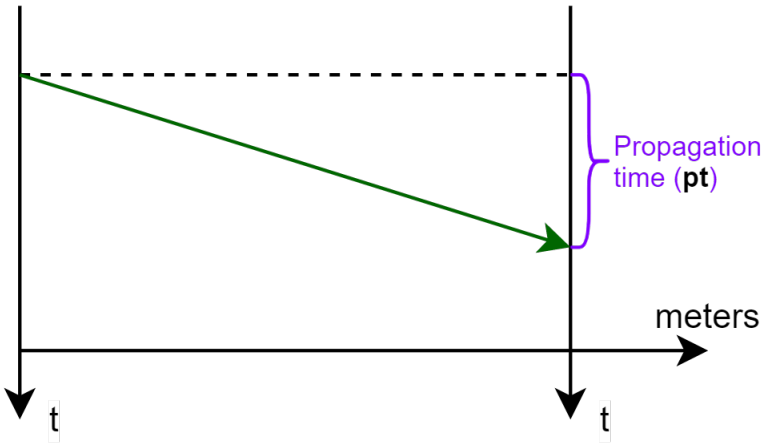


Figure 5.6: Propagation time.

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) $[0, 2^n - 1]$

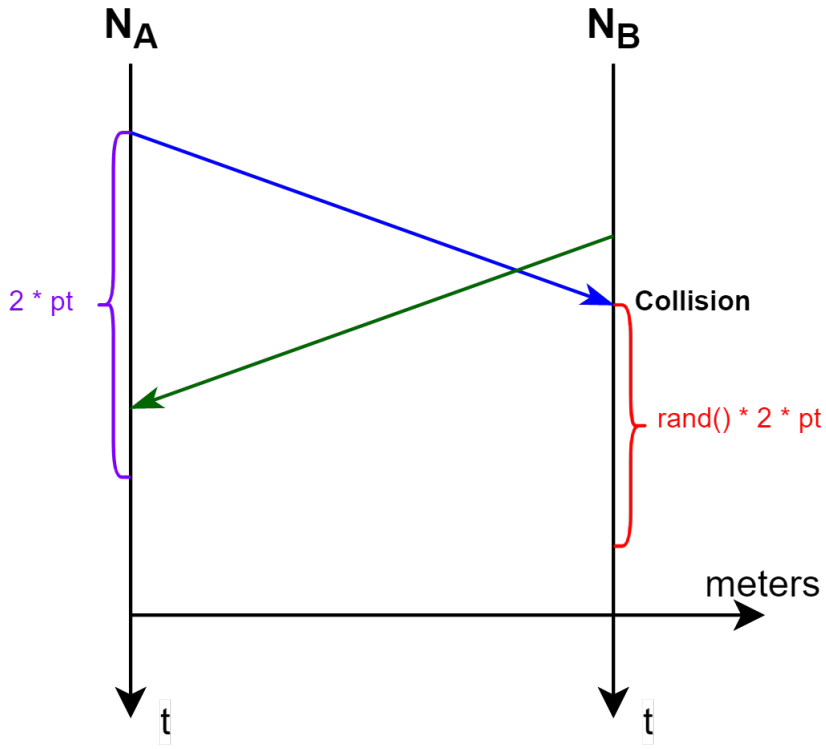


Figure 5.7: Collision avoid.

5.1.1 Ethernet frame

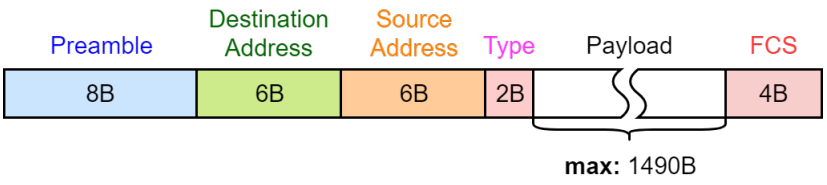


Figure 5.8: Ethernet packet.

- **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

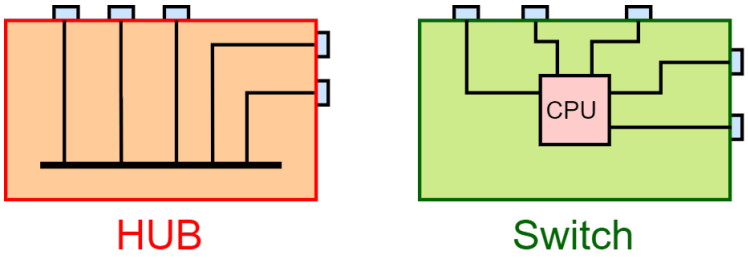


Figure 5.9: Hub and switches.

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

- **Hub**

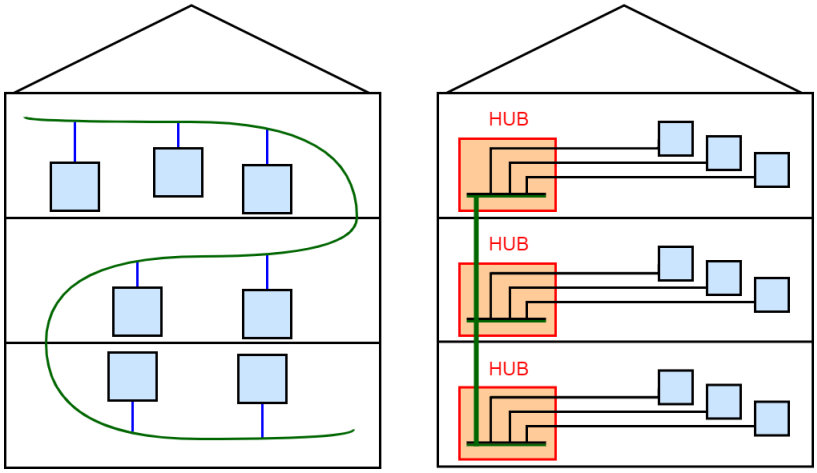


Figure 5.10: Cabled LAN vs LAN with hubs.

  - 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**

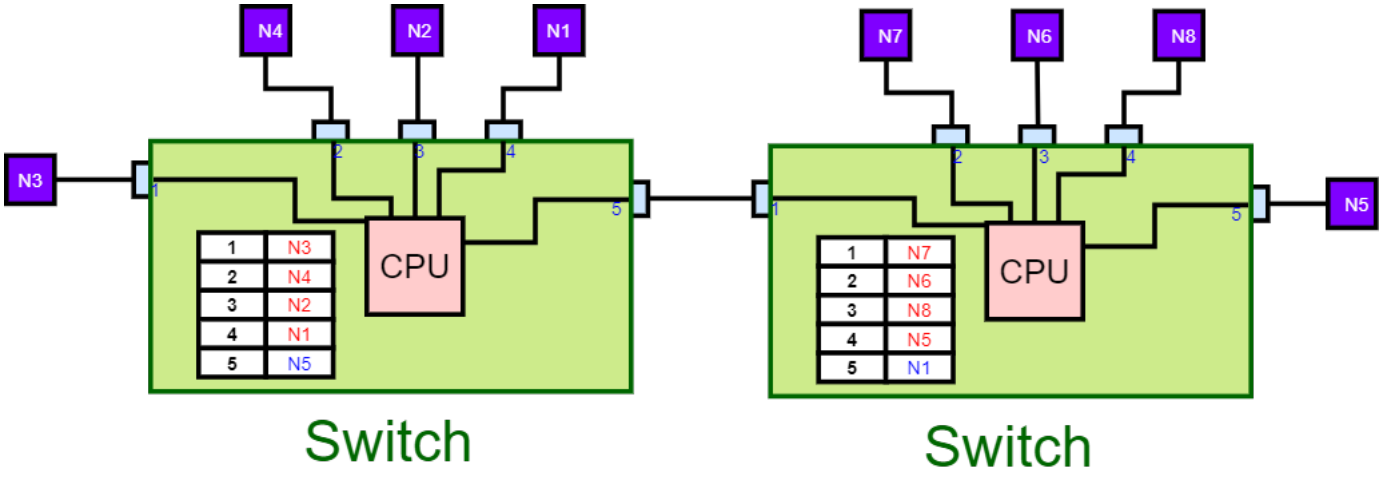


Figure 5.11: Switch connection.

- 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.

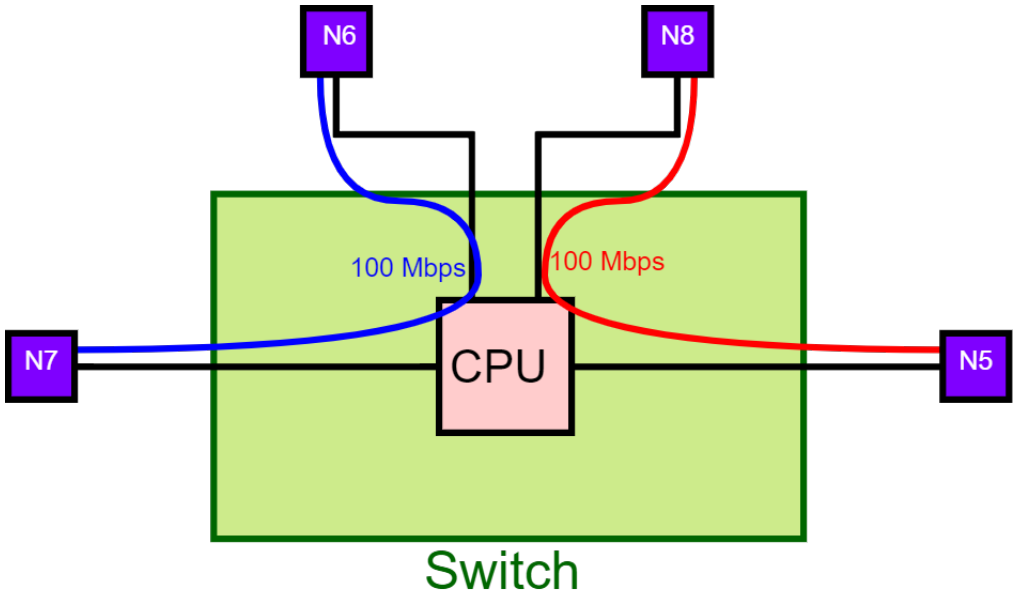


Figure 5.12: Bandwidth switching.

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.

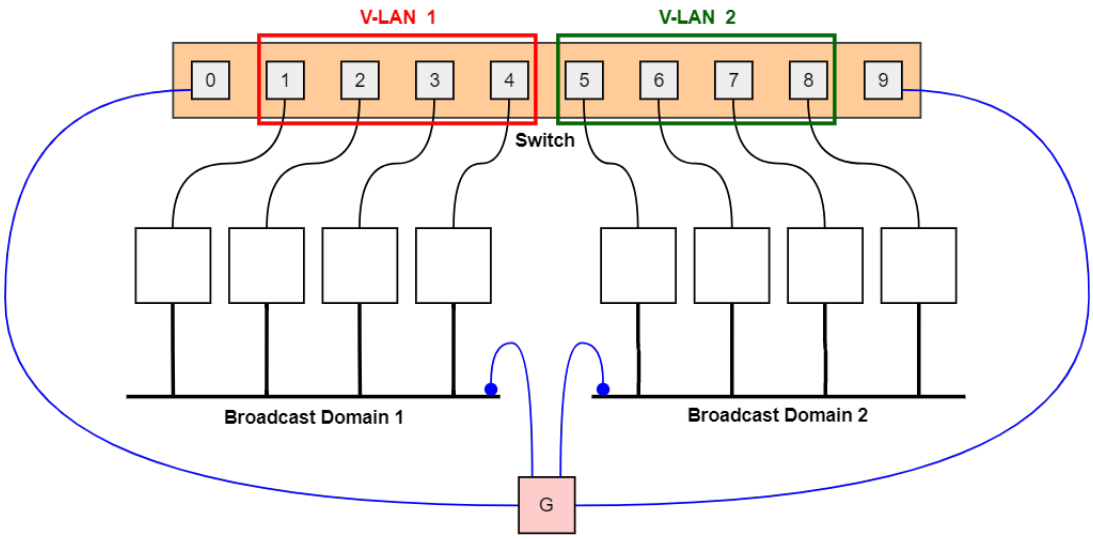


Figure 5.13: VLANs.

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)*.

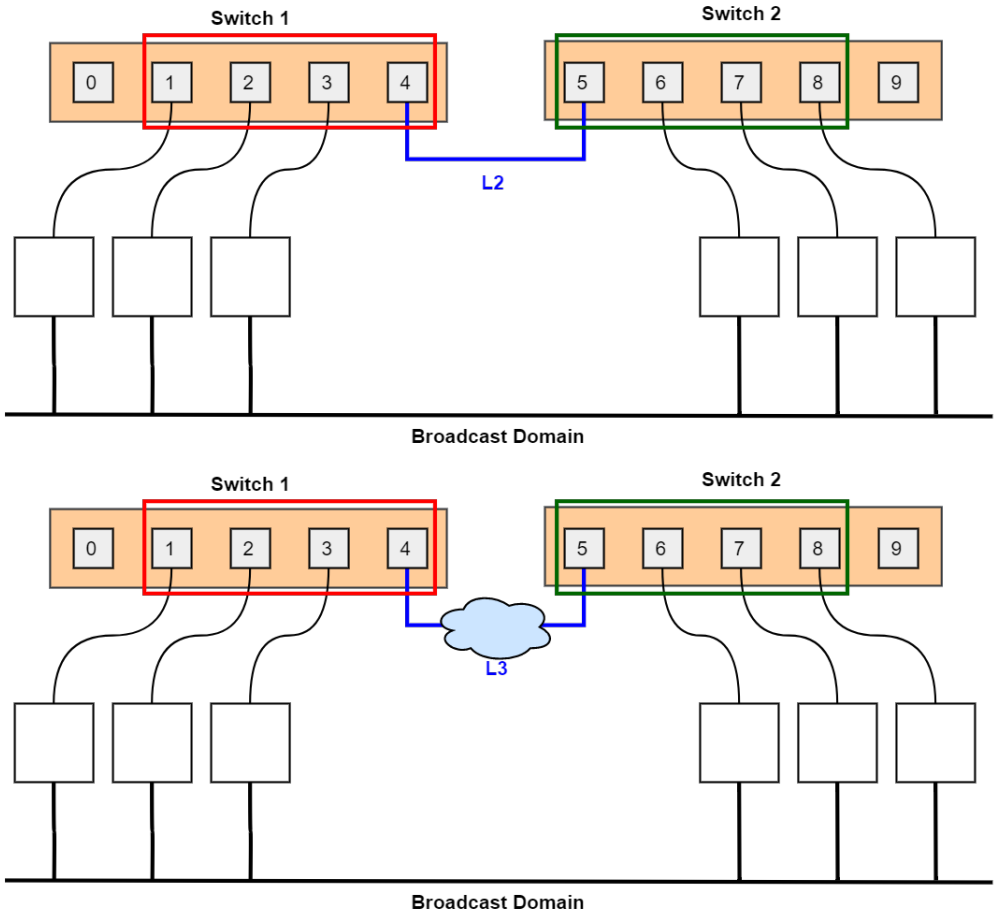


Figure 5.14: VLAN over two switches.

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)
    */
}
```

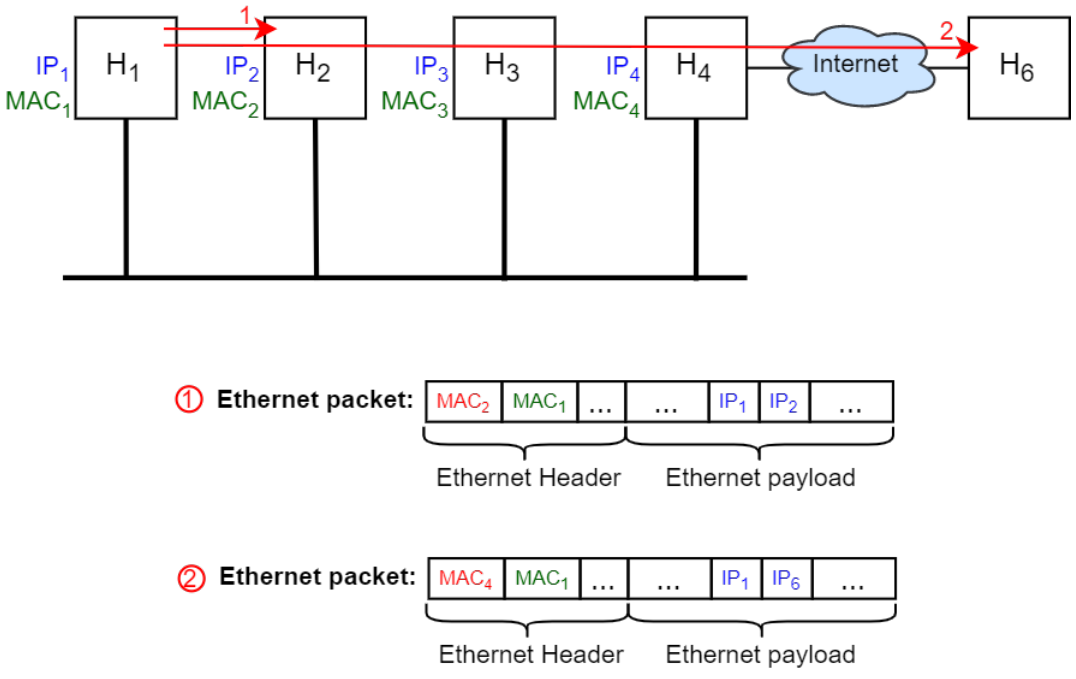


Figure 5.15: ARP.

5.1.4.1 ARP message format

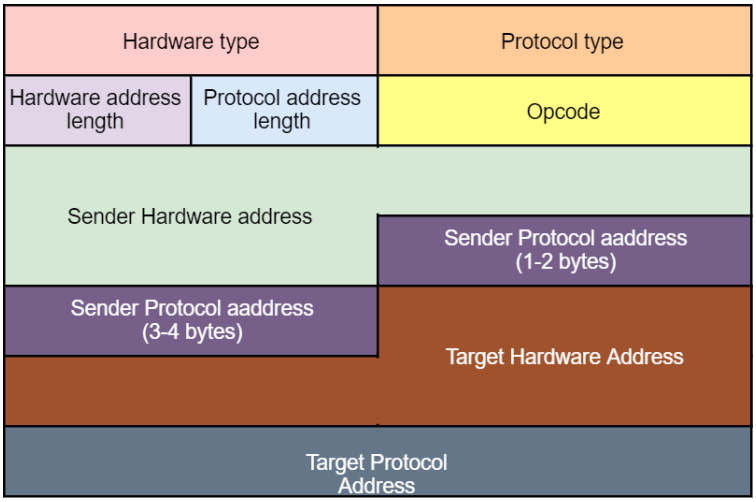


Figure 5.16: ARP message format.

- **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.

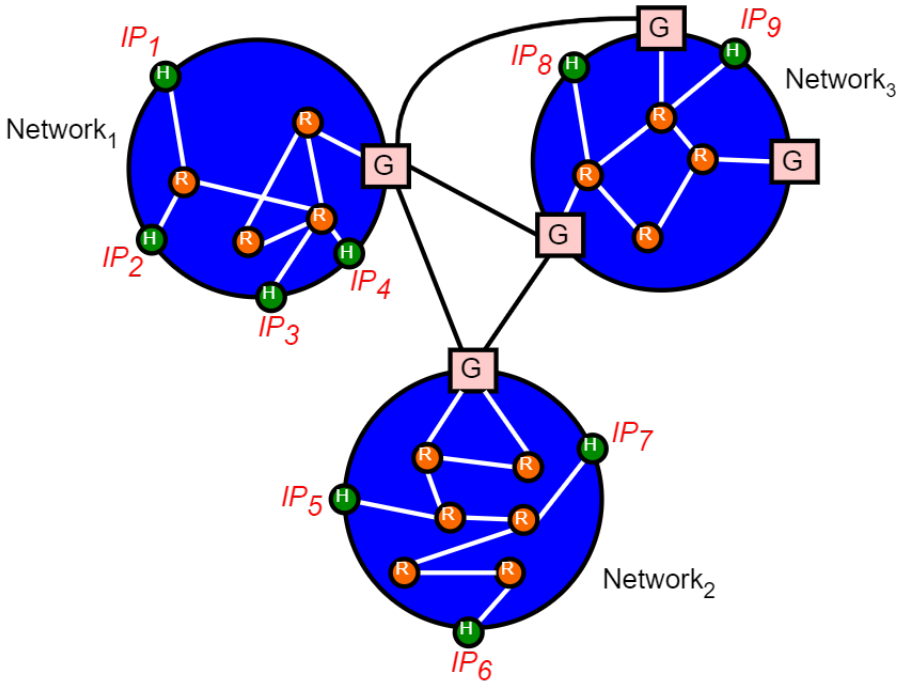


Figure 6.1: Internet structure.

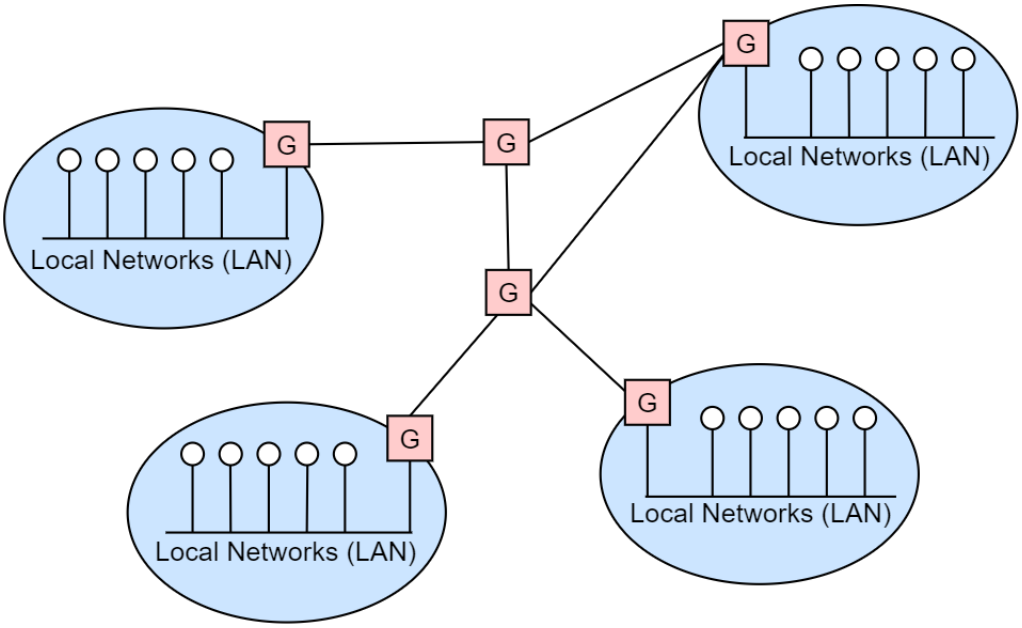


Figure 6.2: LAN structure.

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).

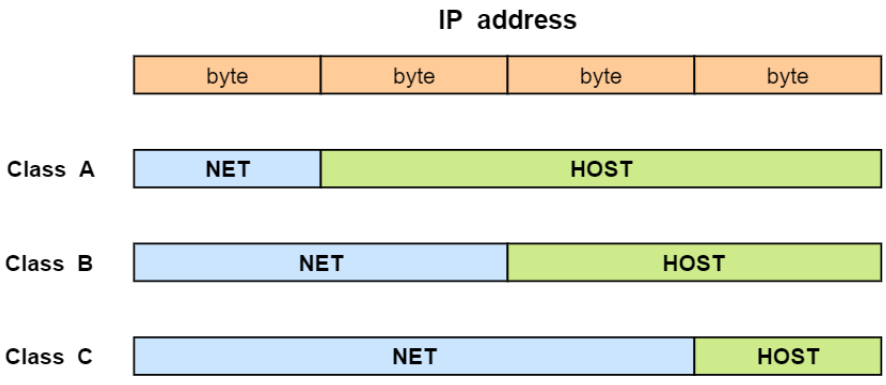


Figure 6.3: IP classes.

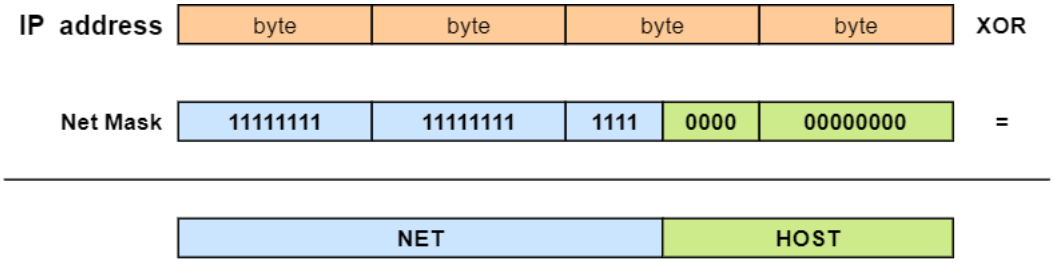


Figure 6.4: Example of netmask use.

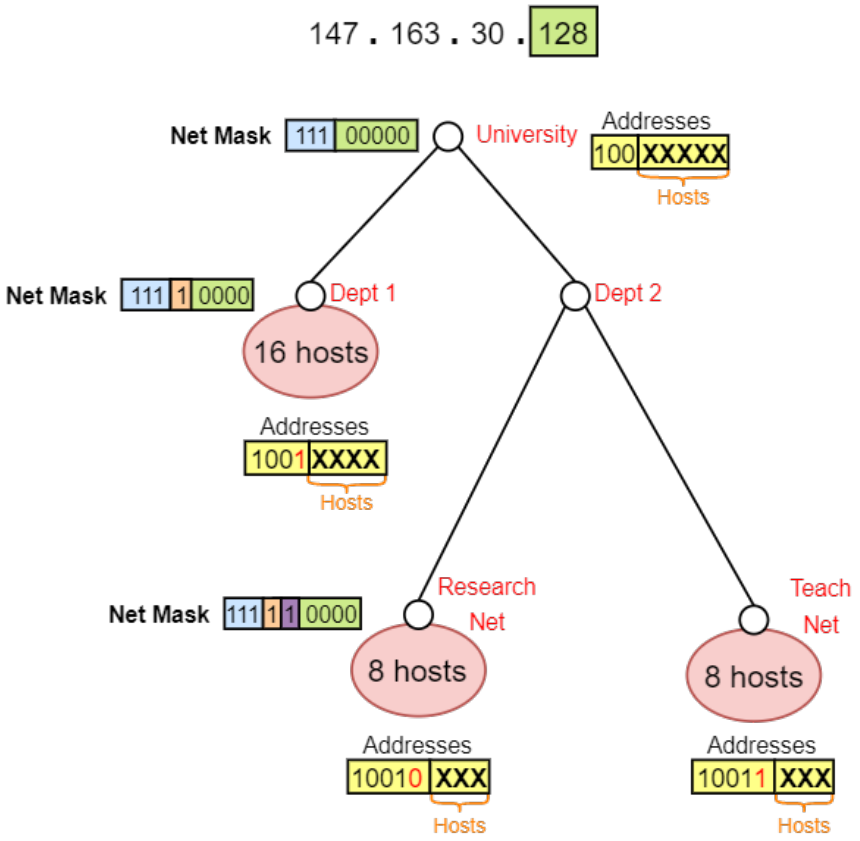


Figure 6.5: Example of subnetworks structure.

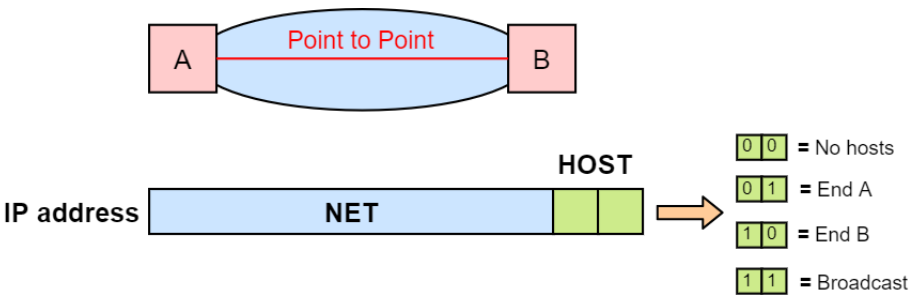


Figure 6.6: Example of Point to Point connection network.

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).

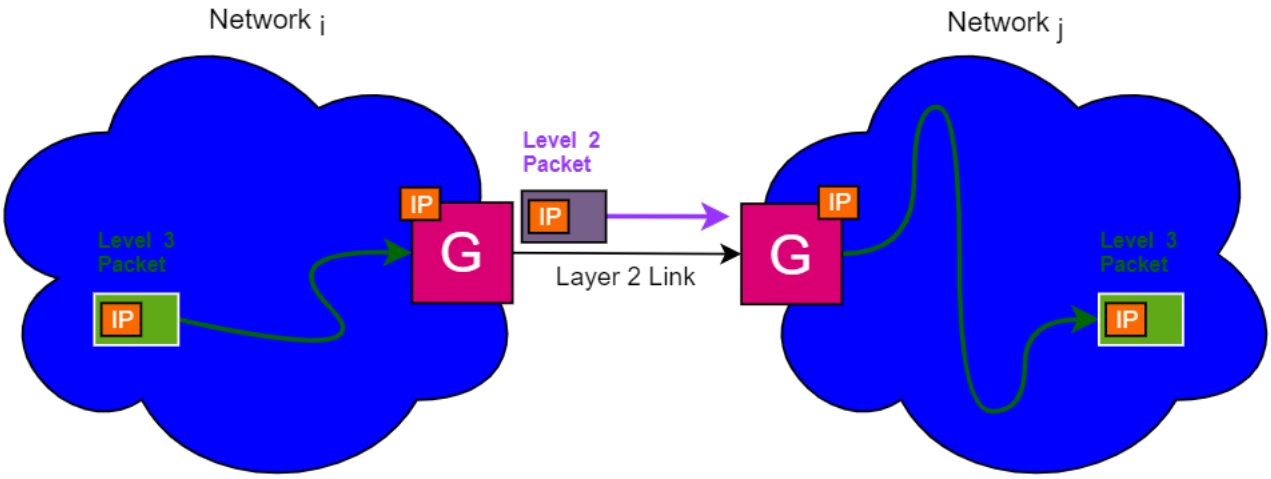


Figure 6.7: Example of encapsulation of IP packet.

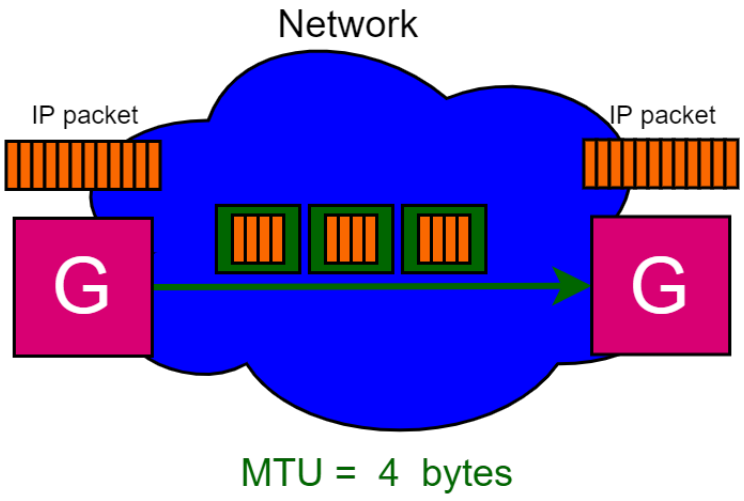


Figure 6.8: Example of fragmentation.

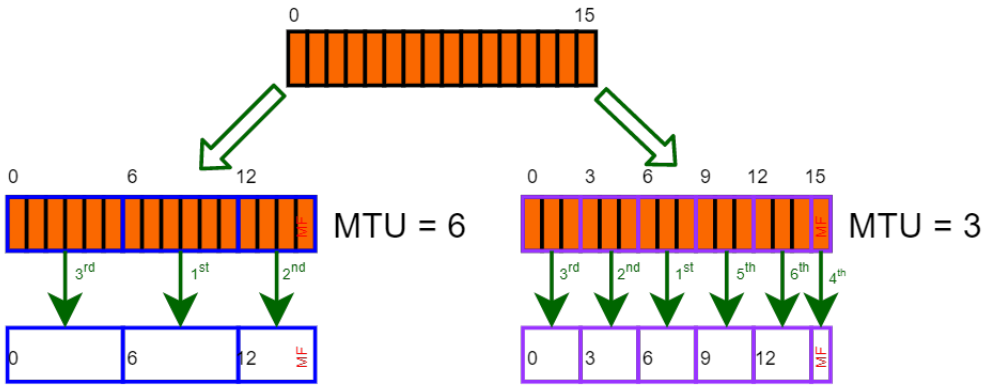


Figure 6.9: Example of fragment labeling.

6.4 Internet Header Format

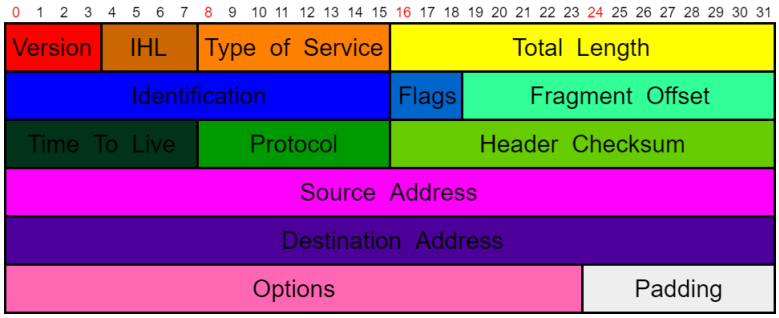


Figure 6.10: Internet header format.

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.

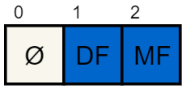


Figure 6.12: Flags.

	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.

- **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.

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

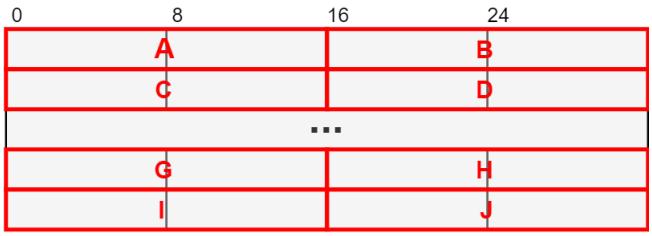


Figure 6.13: Words of payload evaluated in checksum.

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

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.

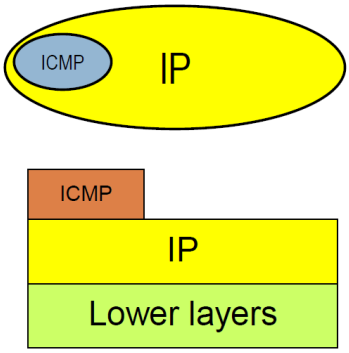


Figure 7.1: How ICMP is embedded in IP datagrams.

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.

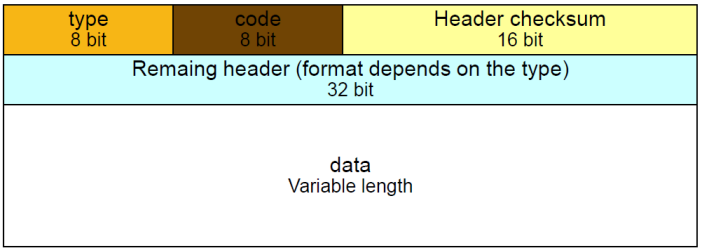


Figure 7.2: Format of ICMP message.

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.

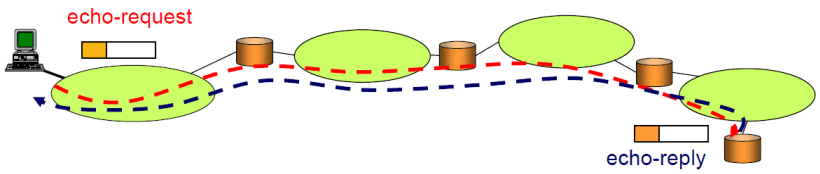


Figure 7.3: ECHO requests and replies in practice.

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

$$\text{Code:} \Rightarrow 0$$

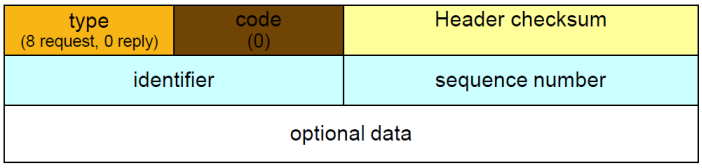


Figure 7.4: Format of ECHO message.

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.

$$\text{Type:} \Rightarrow 3$$

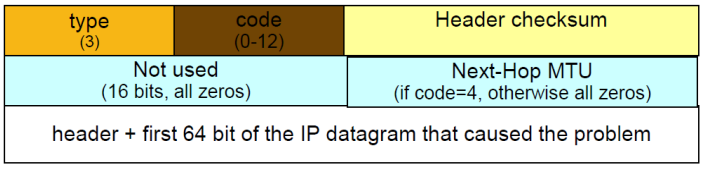


Figure 7.5: Destination unreachable message format.

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: $\Rightarrow 3$

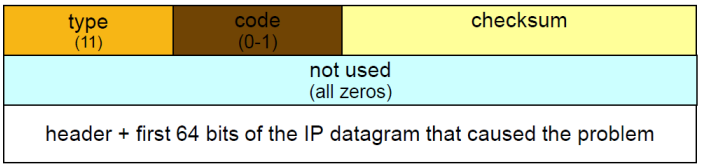


Figure 7.6: Time exceeded message format.

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: $\Rightarrow 12$

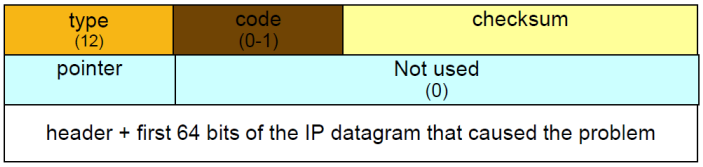


Figure 7.7: Format of Parameter problem message.

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: $\Rightarrow 0-3$

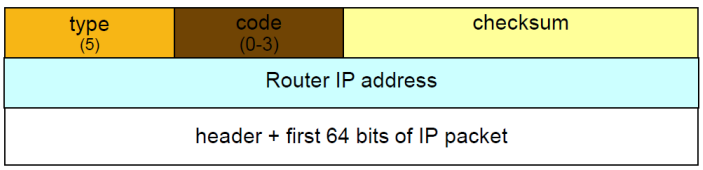


Figure 7.8: Format of Redirect message.

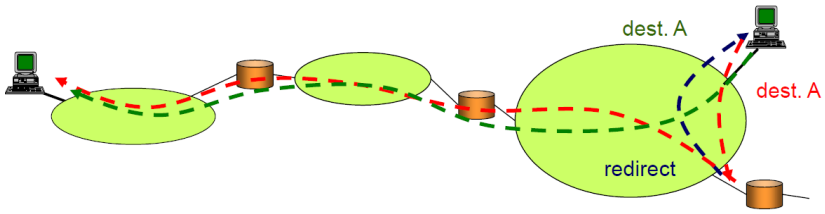


Figure 7.9: How Redirect messages are used.

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

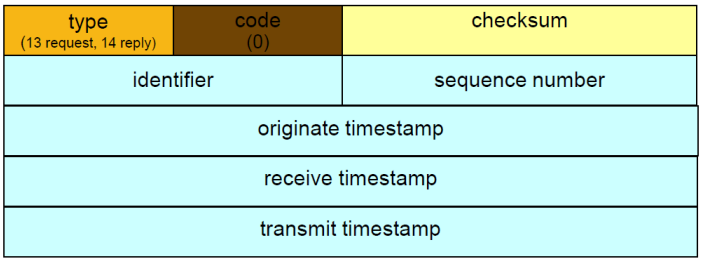


Figure 7.10: Format of Timestamp request and reply.

- **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

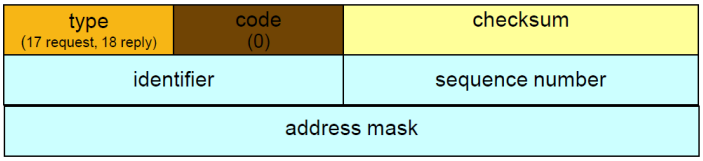


Figure 7.11: Format of Address mask request and reply.

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

Chapter 8

HTTP protocol

HTTP protocol was presented for the first time in the RFC 1945 [1] (Request for Comment).
The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary 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 there was Hypertalk, made by Apple before.
A feature of HTTP is the typing of data representation, allowing systems to be built independently f the data being transferred. HTTP has been in use by the World-Wide Web global information initiative since 1990.

8.1 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.

8.2 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)>

8.3 Messages

8.3.1 Different versions of HTTP protocol

- **HTTP/0.9 Messages**
  Simple-Request and Simple-Response do not 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]
```

8.3.2 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
```

8.3.3 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.

8.3.4 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.

8.3.5 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
```

8.3.6 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

8.4 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

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

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.

8.4.1 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)

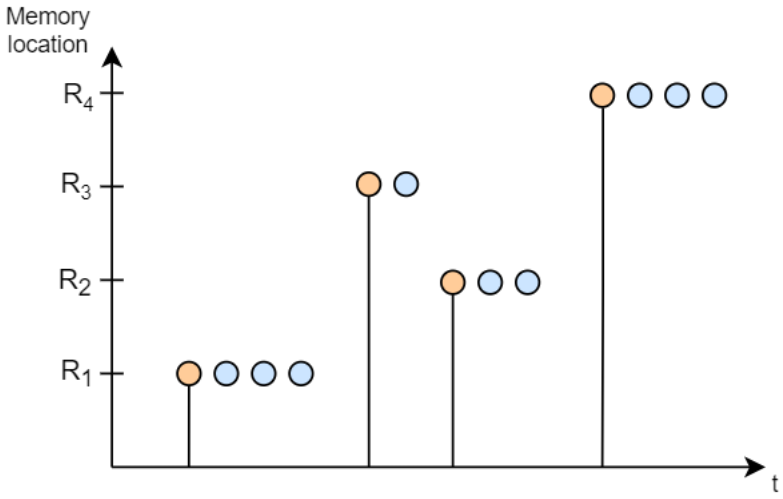
- **Location**
  defines the exact location of the resource that was identified by the Request-URI (useful for 3xx responses)
- **Pragma**
  It's sent by server to inform that there in no caching systems
- **Referer**
  allows the client to specify, for the server's benefit, the address (URI) of the 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.

8.4.2 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.
The update of the content of the local cache for the client can happen in three different ways:

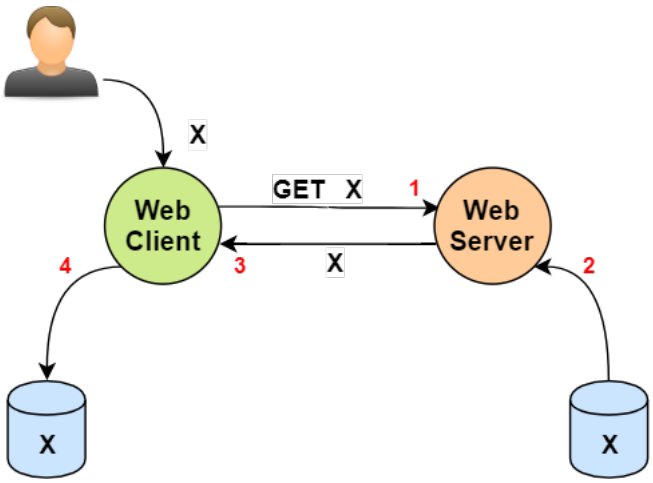
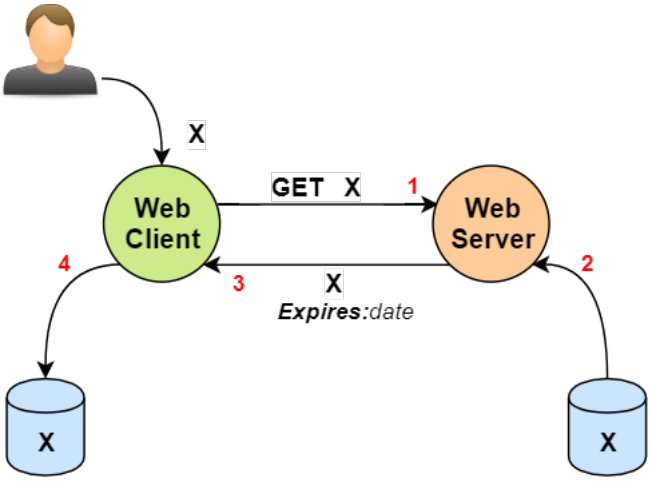


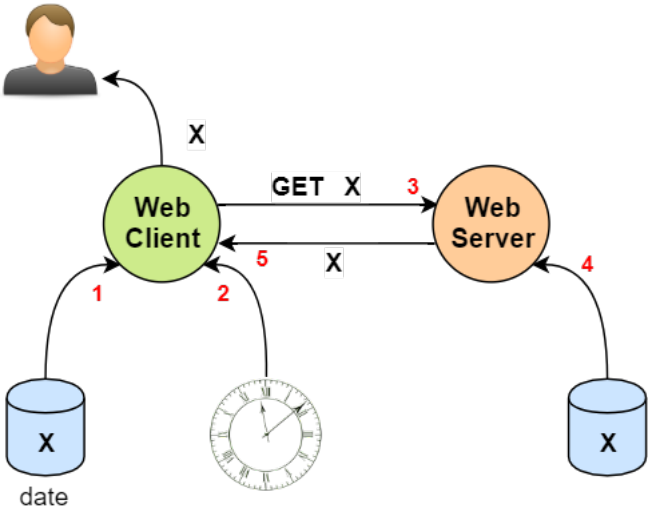
Figure 8.1: First insertion of the resource in the cache.

- **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.

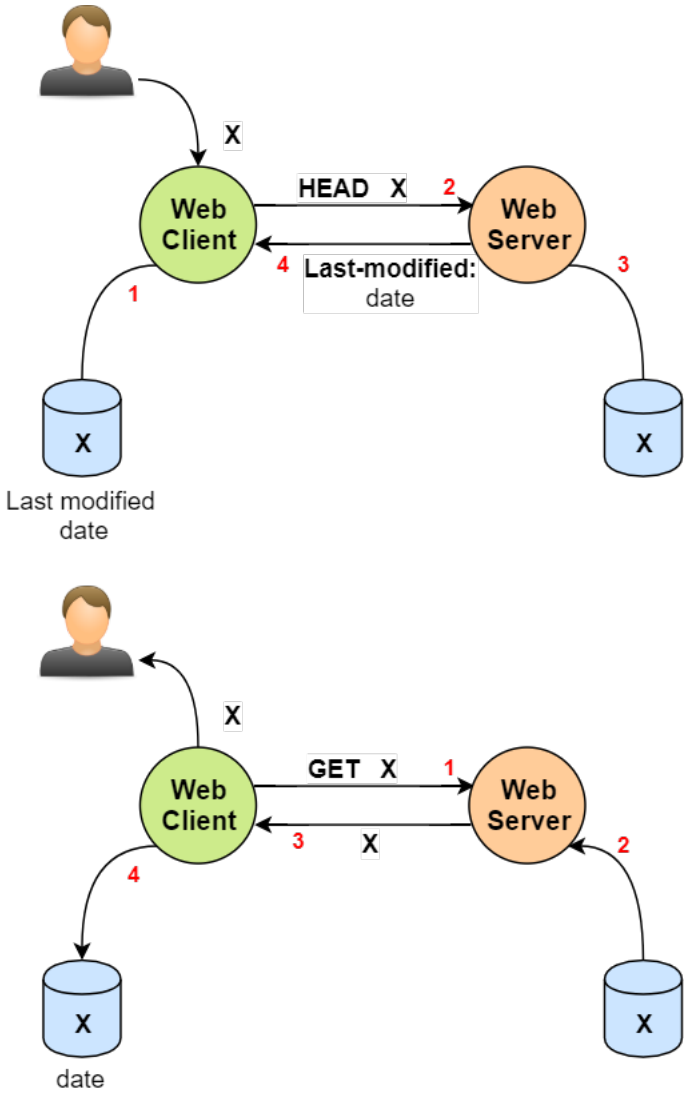

X
Web Client
GET X
1
Web Server
4
3
X
Expires:date
2
X
X


X
Web Client
GET X
3
Web Server
5
X
1
2
4
X
date
X

- **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.

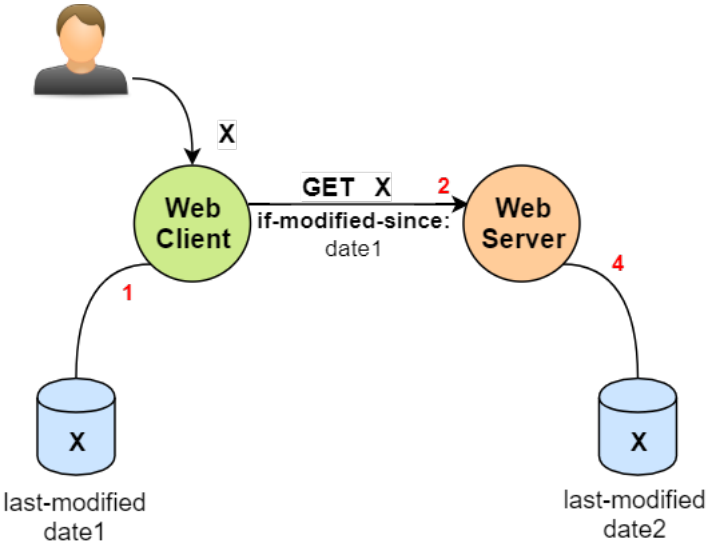
  The problem of this method is that, in the worst case, we send two times the request of the same resource (even if the first one, with "HEAD" method, is less heavy).

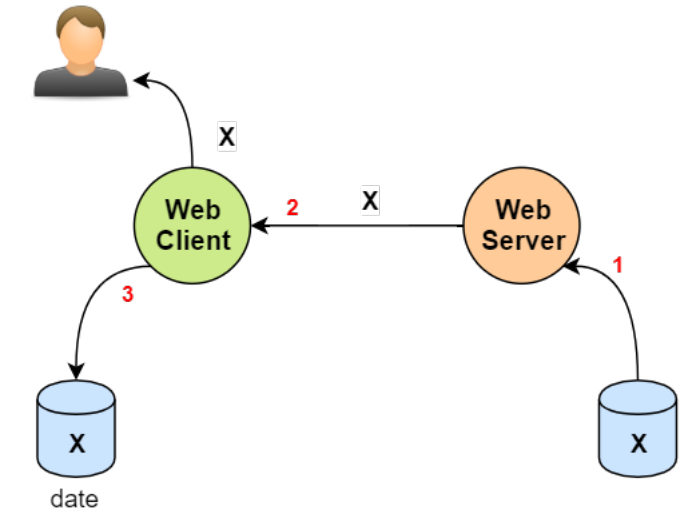


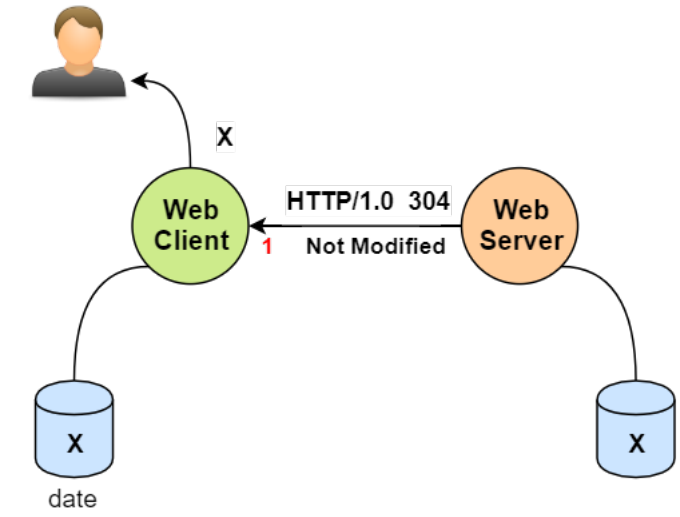
- **Request with if-modified-since header**

  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 correpт answer without other requests.


X
Web Client
GET X
if-modified-since:
date1
2
Web Server
4
1
X
last-modified
date1
X
last-modified
date2


X
Web Client
2
X
Web Server
1
3
X
date
X


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

8.4.3 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:

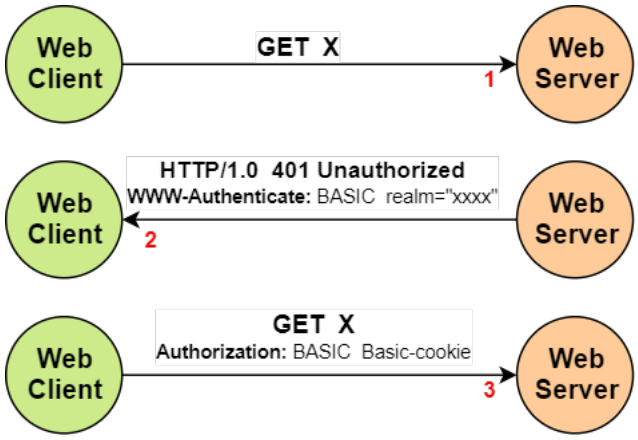
```
WWW-Authenticate: Auth-Scheme Realm="XXXX"
```

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:

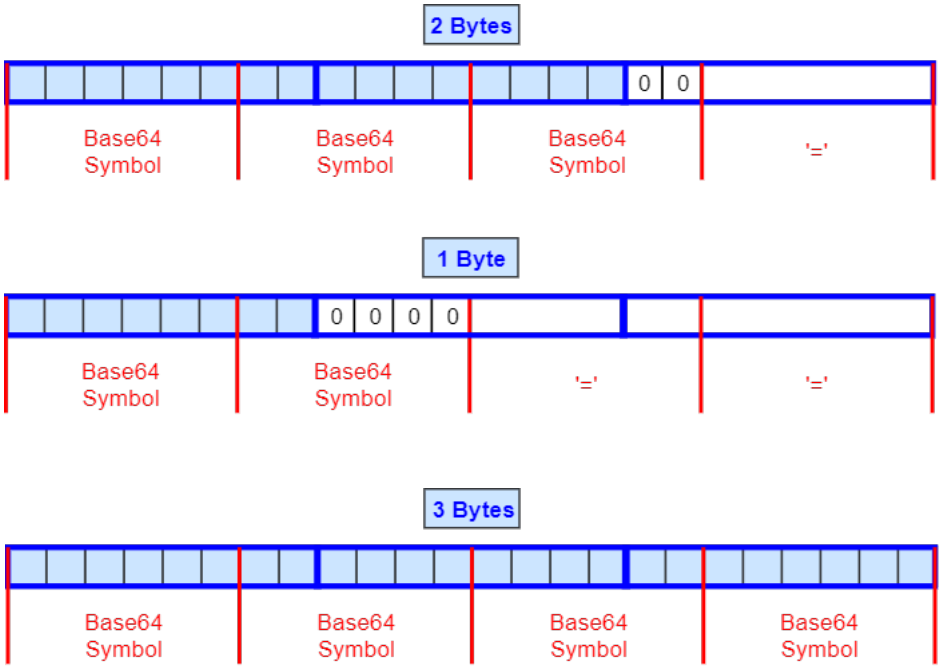
```
WWW-Authenticate: Auth-Scheme Basic-cookie
```

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)



8.4.3.1 base64

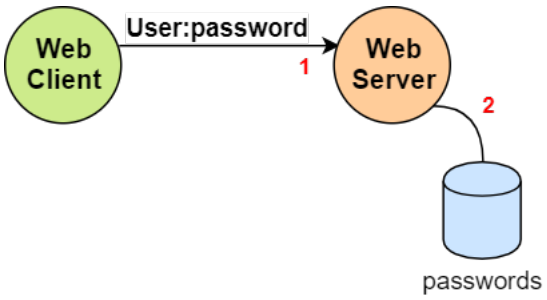
It is very useful for a lot of protocol like HTTP, that doesn't support format different than text of characters. For example with SMTP, all the mail contents must be text, hence images or other binary files are encrypted with base64.
Starting from a stream of bytes, we are going to convert numbers, 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.



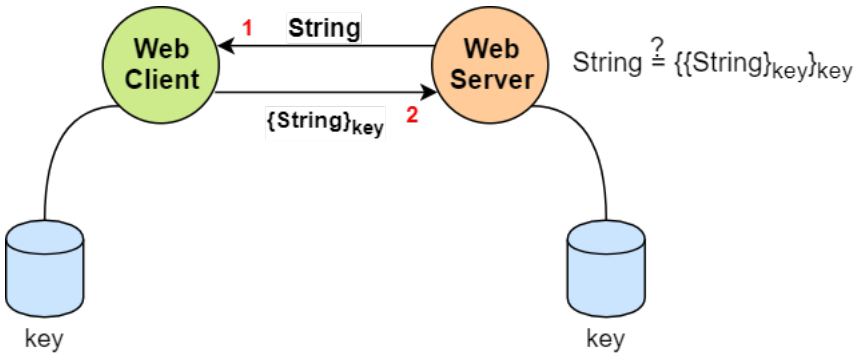
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47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

8.4.3.2 Auth-schemes

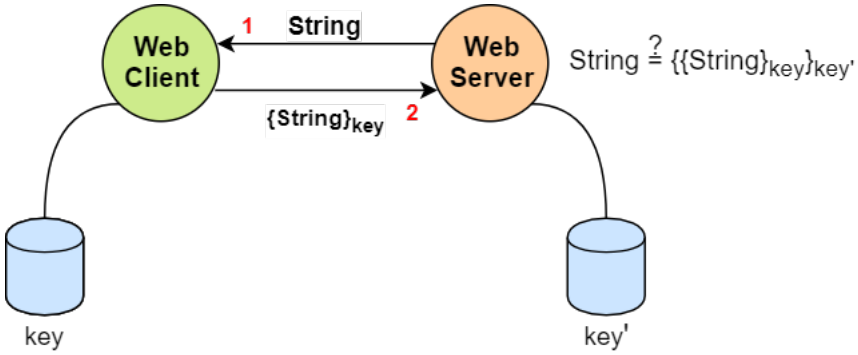
- **BASIC**



- **Challenge** (symmetric version)



- **Challenge** (asymmetric version)



8.5 HTTP 1.1

The architecture of the model is in RFC2616 [2]. 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 8.2). 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)
```

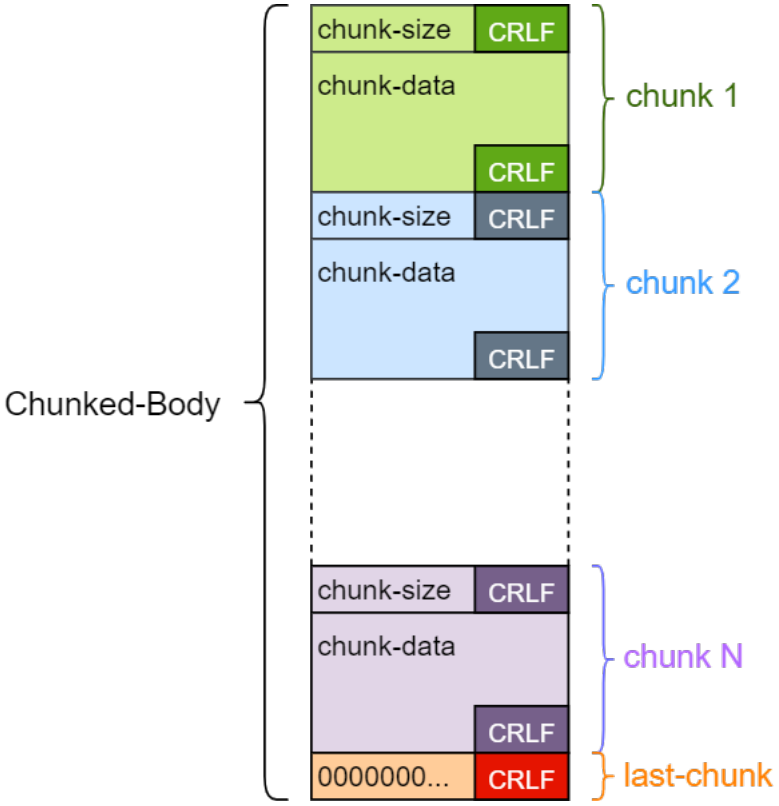


Figure 8.2: Chunked body.

8.5.1 Caching based on HASH

It's like the caching mechanism used looking to "Last-Modified" header value through the use of HEAD. 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.

8.5.2 URI

In URI, there is the encapsulation of the operation done in the past, to have a resource from a server [3]. 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,
```

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

8.5.3 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 [4], for example **ftp** to download resources.

8.6 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 8.3).
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

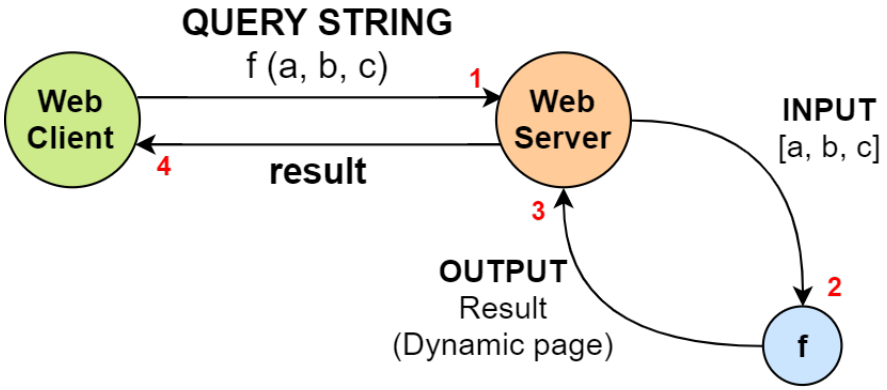


Figure 8.3: Example of CGI application.

8.7 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.

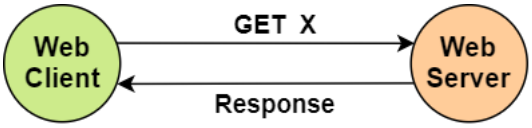


Figure 8.4: Direct access.

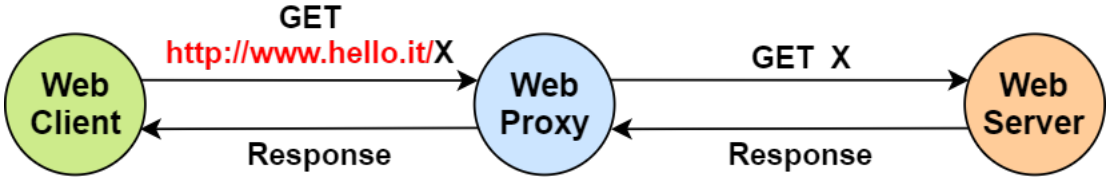


Figure 8.5: Proxy access.

- **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.

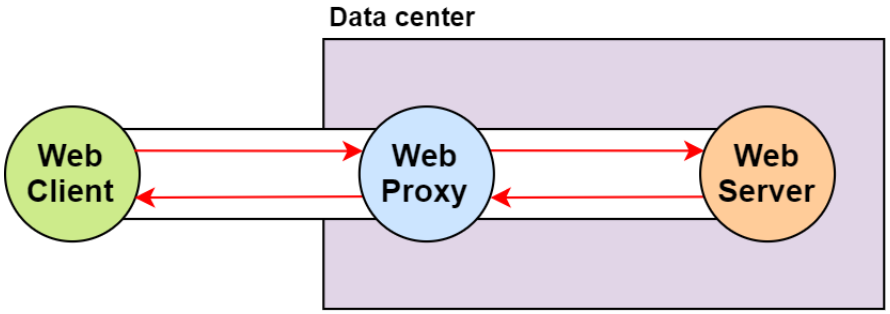


Figure 8.6: Proxy as WAF in HTTPS.

– **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 [5].
In HTTPS the client uses the method *CONNECT* to tell to the proxy to work as a tunnel. The proxy, receiving the *CONNECT* request, extablish 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 8.9):

  * **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.

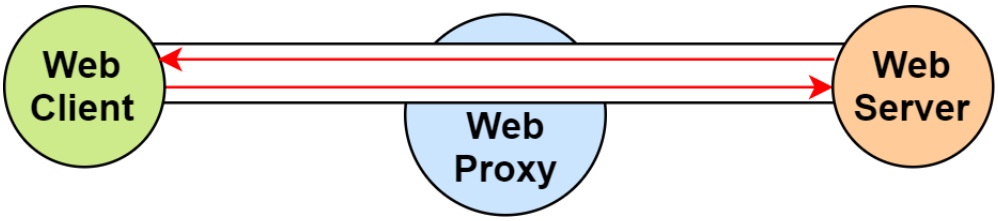


Figure 8.7: Tunneling using proxy in HTTPS.

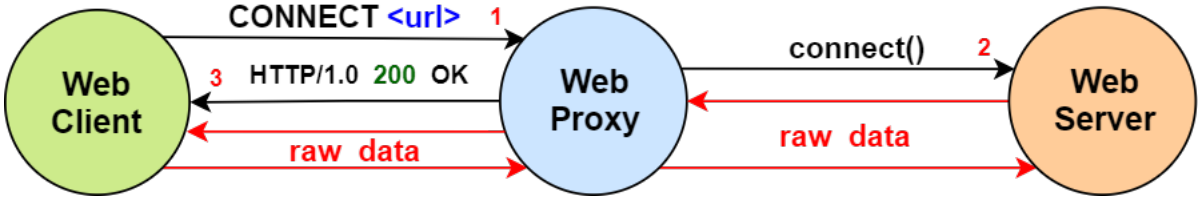


Figure 8.8: CONNECT request in HTTPS.

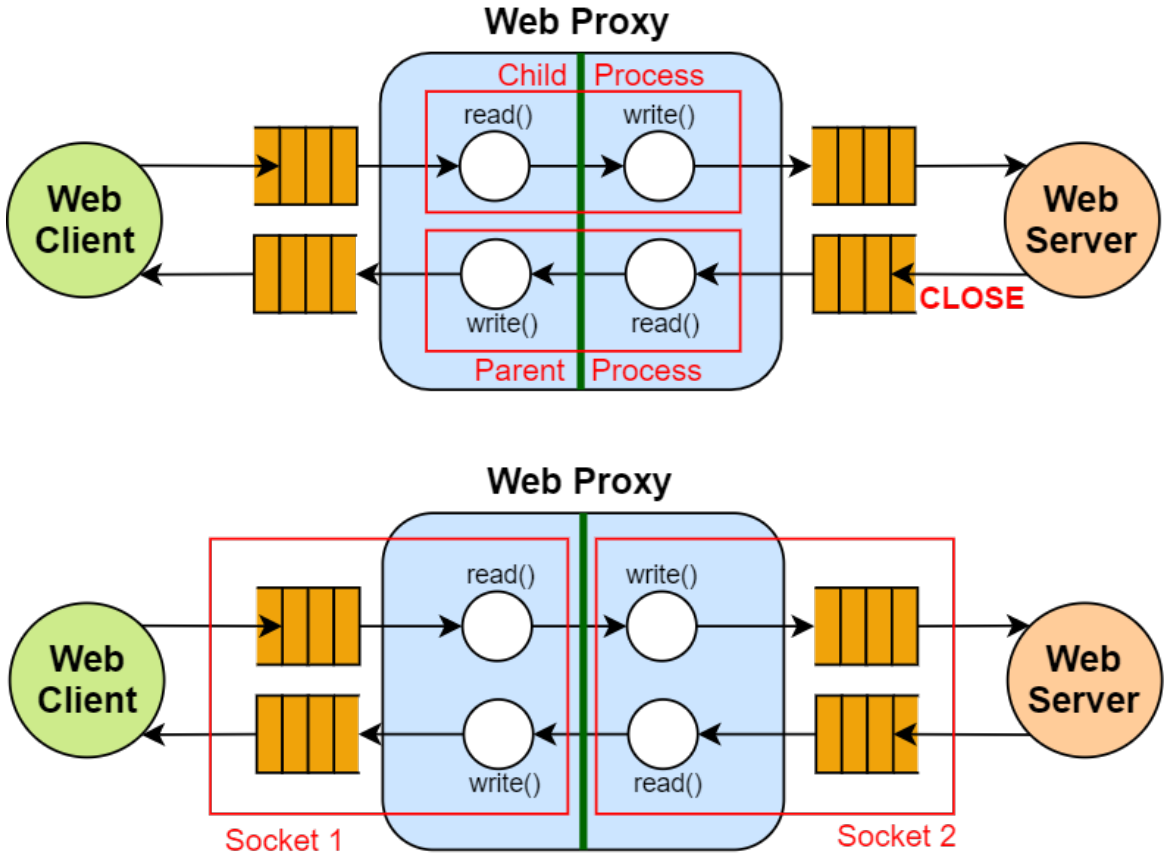


Figure 8.9: How proxy works with HTTPS.

Chapter 9

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.

9.1 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 9.1).
This system collects all the files, like an hub, and shared resolution names to other clients (Figure 9.1).
This architecture is unfeasable and not scalable with nowadays number of IP addresses, because the files become very huge and transfering becomes very slow.

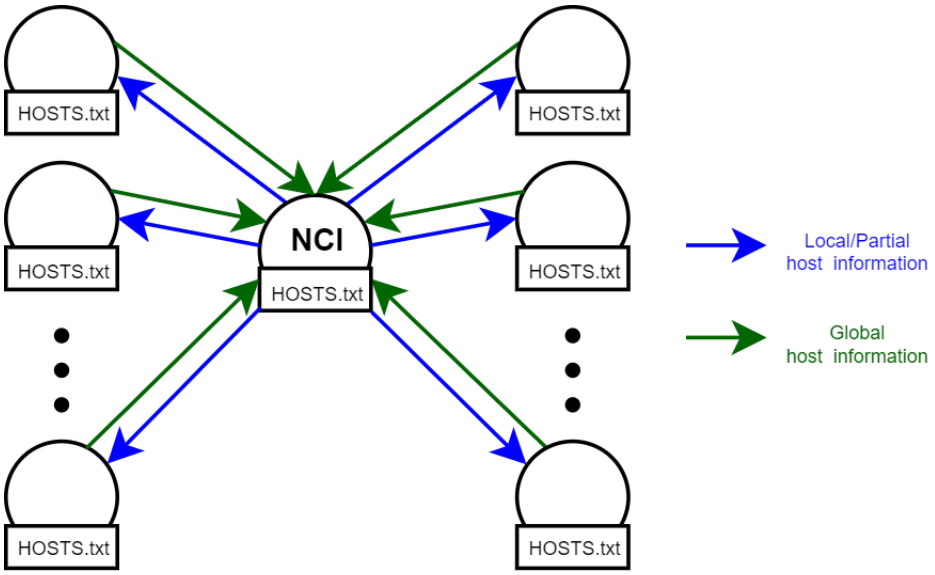


Figure 9.1: How NIC worked.

9.2 Domain Name System (DNS)

The file **HOSTS.txt**9.1 is yet used in nowadays UNIX systems. 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 [6].

9.2.1 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.

9.2.2 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 9.2).
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 needs us to understand how to rich 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 also in DNS but at level 4. There are 13 root name servers that are returned 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 transfering. The local DNS server for the device, managed by my network provides, contains the 13 root servers and permit us to reach at least one DNS root server.
The 13 DNS root servers are added locally at its installation of local DNS 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 domain resolution for a domain that composes name 9.3.
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 **??**. All the name servers, where the query passes throuh, store information about resolution. This system is never applied as it is.

In reality we use an hybrid version, that uses partially recursion ??. 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 yes and that is used in UDP packet as an additional information. The Root Name Server decides if it wants to accept recursive query or if not, how mmany domains can resolve.
I can group some domains, defining a zone, so I can use only a name server for the zone that solves more domains together 9.6. So the name servers are authorithative over zones and not only single domains. It depends if domains are grouped or not.
The creation of the zones are used to manage easily the responsability and the organization over the zones, partitioning them. Another reason for this partition in zones is because of some domains has few names and so it's better to group the domain with others.

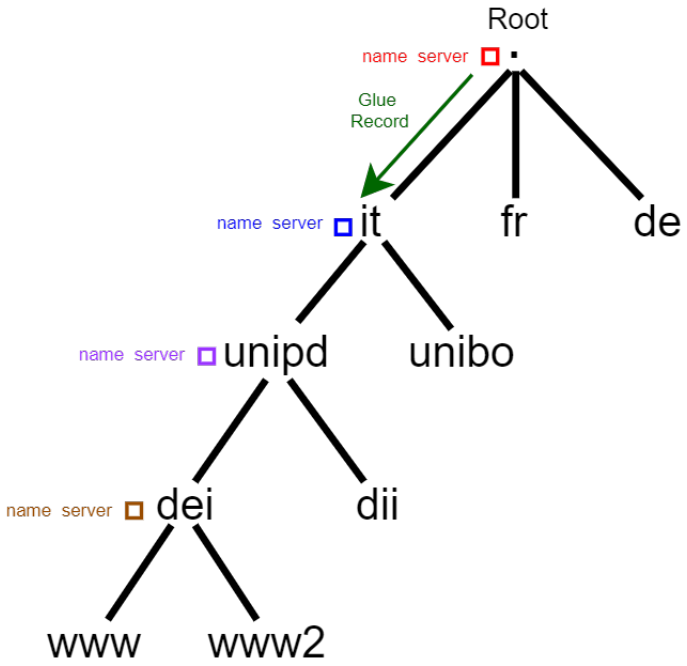


Figure 9.2: DNS structure.

```
//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
```

Listing 9.1: Example of default DNS queries using dig.

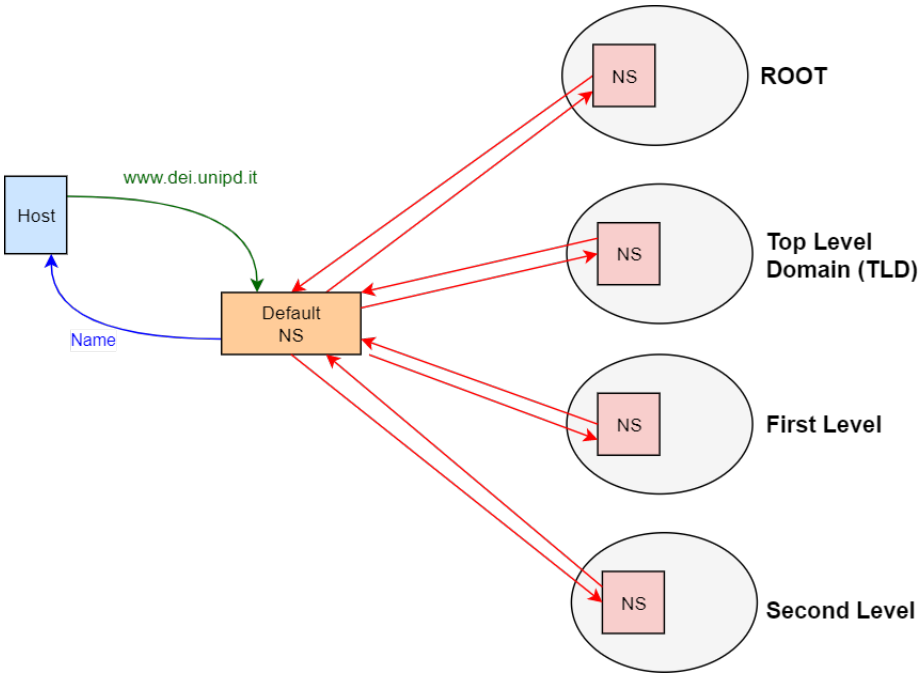


Figure 9.3: Default DNS behaviour without caching.

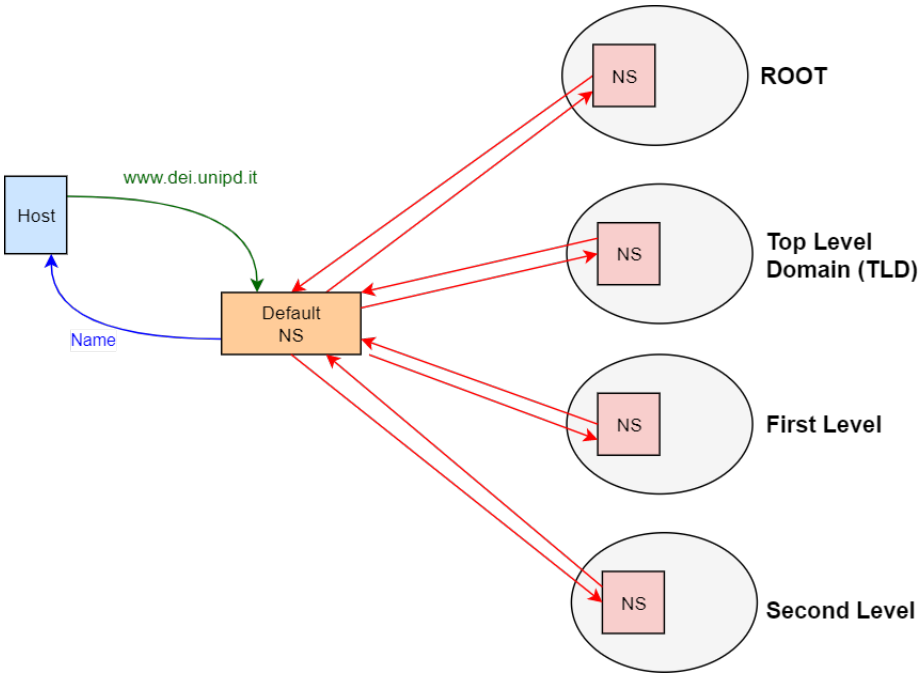


Figure 9.4: Completely recursive DNS structure.

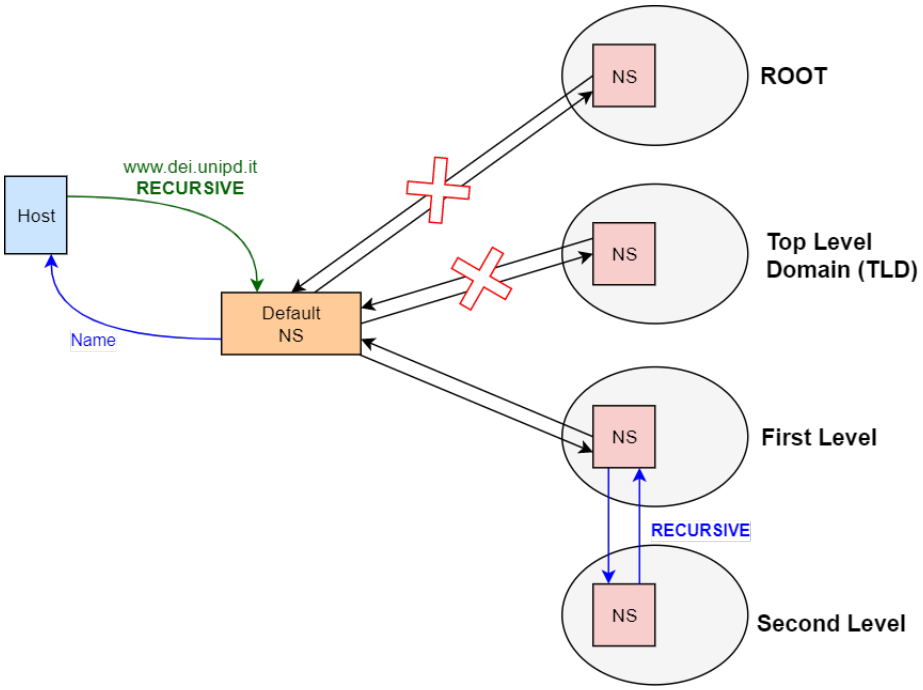


Figure 9.5: Hybrid DNS structure.

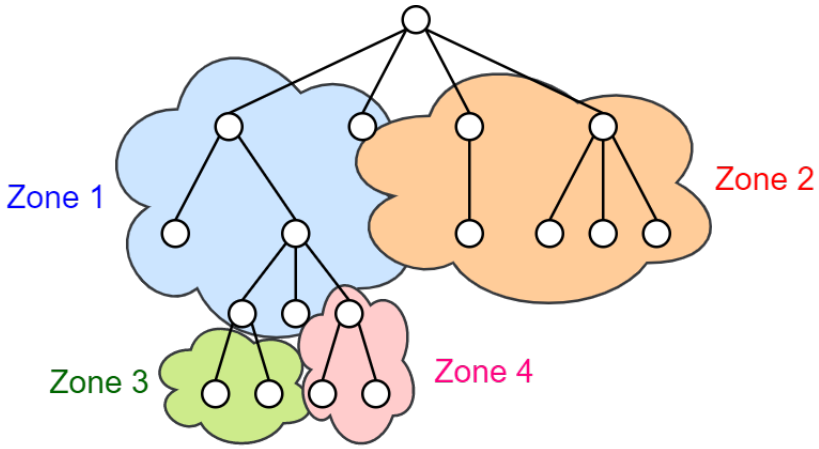


Figure 9.6: Example of partitioning into zones.

Chapter 10

Shell

10.1 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.
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 **arp** -a		Show numerical addresses instead of trying to determine symbolic hostnames in routing table. List all the MAC addresses stored after some ARP requests and replies made by our ethernet interfaces.

10.2 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.

Chapter 11

vim

11.1 .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.

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

Listing 11.1: .vimrc

11.2 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

11.3 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]	

References

[1] https://tools.ietf.org/html/rfc1945.

[2] https://tools.ietf.org/html/rfc2616.

[3] https://tools.ietf.org/html/rfc3986.

[4] https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml.

[5] https://tools.ietf.org/html/rfc2817.

[6] https://tools.ietf.org/html/rfc1034.