Padua University

Engineering Course

Master of Computer Engineering

Computer Networks

Raffaele Di Nardo Di Maio

Accademic Year 2019-2020

Contents

Chapter 1

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.

1.1 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.

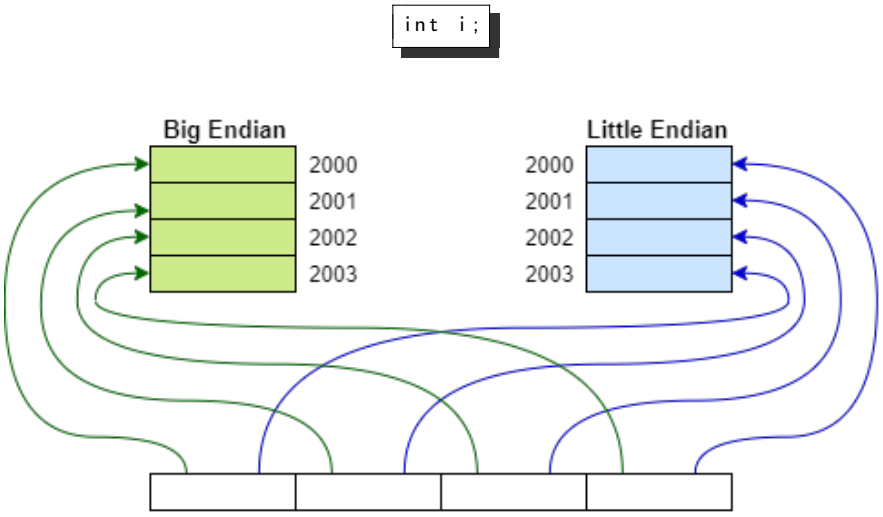


Figure 1.1: Little Endian and Big Endian.

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).

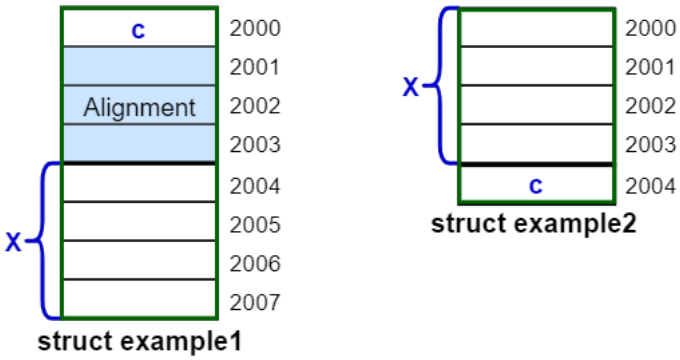
signed	unsigned
int8_t	uint8_t
int16_t	uint16_t
int32_t	uint32_t
int64_t	uint64_t

Table 1.1: <stdint.h>

1.2 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 1.2). 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;
}                        }
```



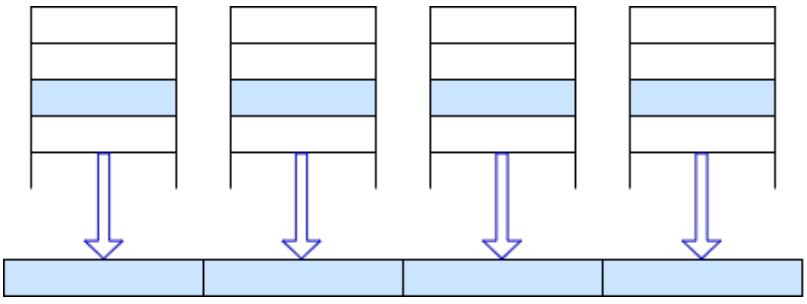


Figure 1.2: Parallel reading in one istruction in 32 bits architecture.

1.3 Structure of C program

The program stores the variable in different section (Figure 1.3):

- **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

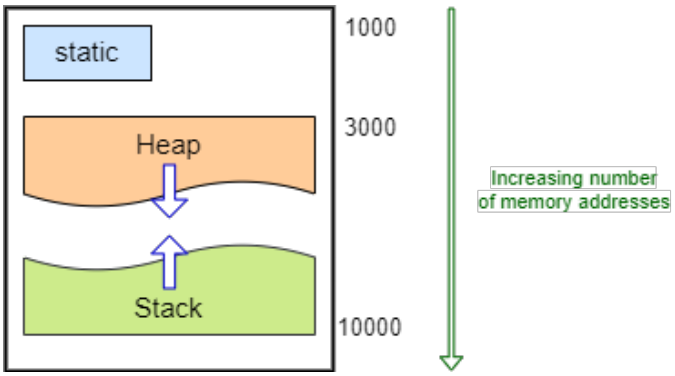


Figure 1.3: Structure of the program.

Chapter 2

Network services in C

2.1 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 2.1). Hence Application and Transport can talk each other with System Calls.

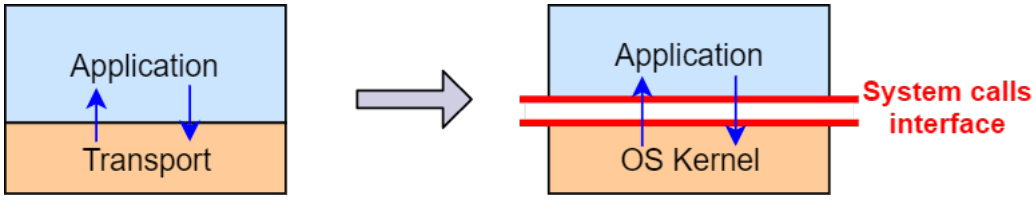


Figure 2.1: System calls interface.

2.2 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*

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

2.3 TCP connection

In TCP connection, defined by type **SOCK_STREAM** as written in the Section 2.2, 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.

2.3.1 Client

2.3.1.1 connect()

The client calls **connect()** function, after **socket()** function of Section 2.2. 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 2.2):

- **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 **??** to find which file contains well known port numbers). *It's unique in the system.*

As mentioned in Section 1.1, 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 as in previous example or with the follow function:

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

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

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

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

uint16_t htons(uint16_t hostshort);
```

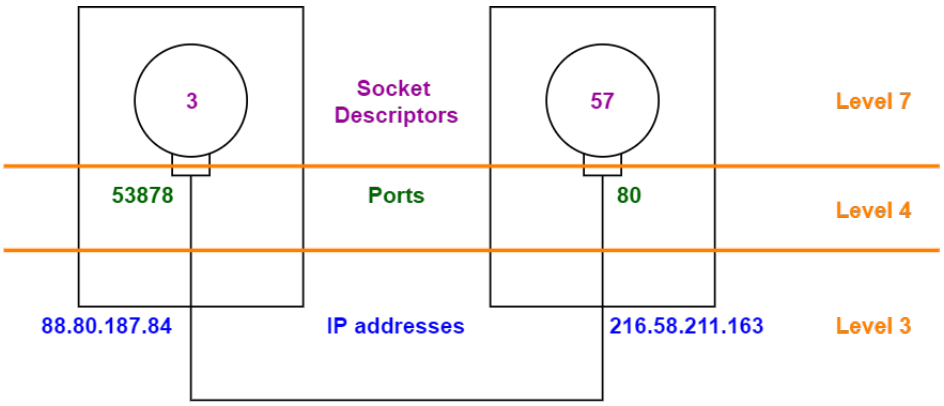


Figure 2.2: After successful connection.

2.3.1.2 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);
```

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).

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.

2.3.1.3 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 it2.2.
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.

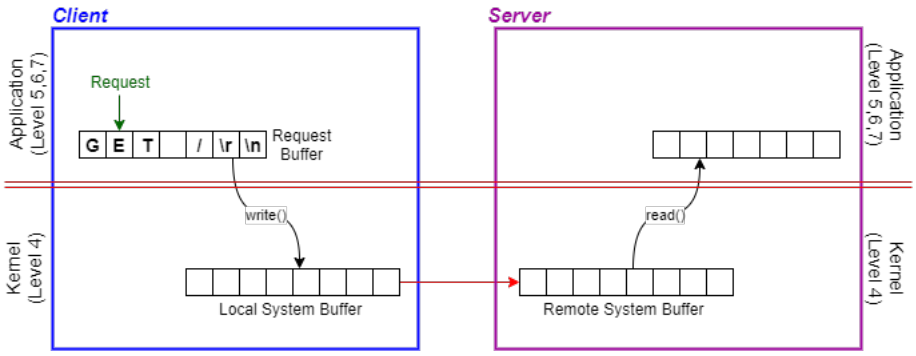


Figure 2.3: Request by the client.

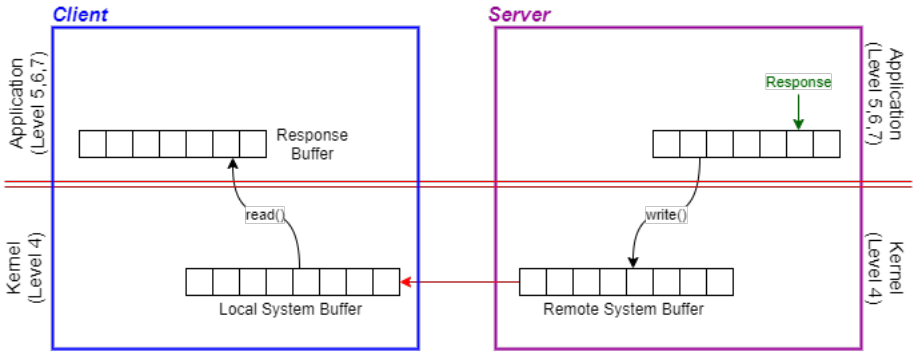


Figure 2.4: Response from the server.

2.3.2 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.

2.3.2.1 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.*

2.3.2.2 listen()

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

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

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 2.5.

The listening socket, identified by **sockfd**, is unique for each association of a port number and a IP address of the server (Figure 2.6).

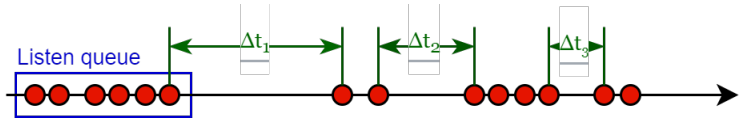


Figure 2.5: Poisson distribution of connections by clients.

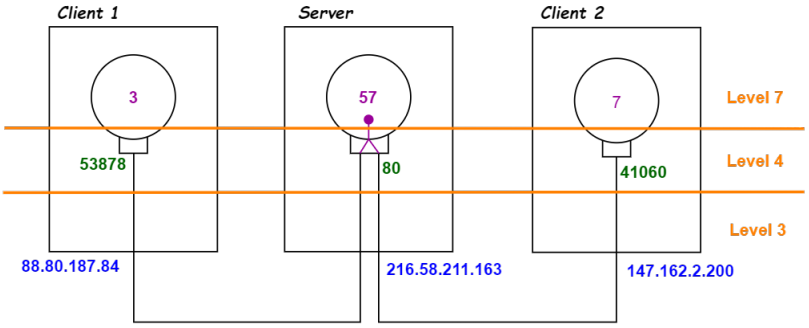


Figure 2.6: listen() function.

2.3.2.3 accept()

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

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

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 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 2.8).
At lower layers of ISO/OSI, the port number and the IP Address are the same identifiers, to which listening socket is associated (Figure 2.7).

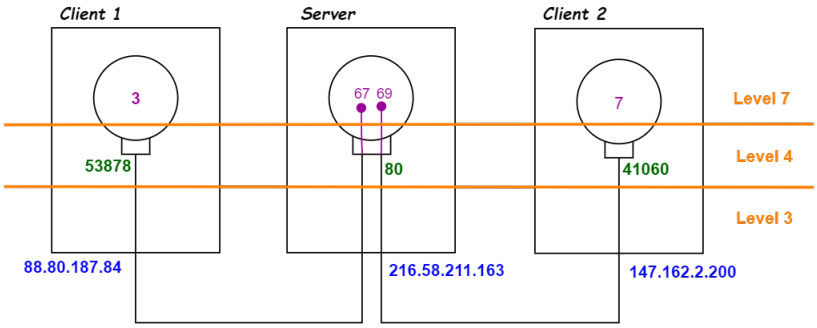


Figure 2.7: accept() function.

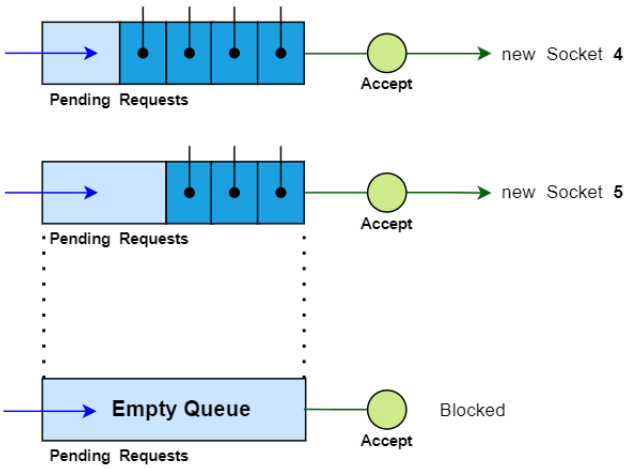


Figure 2.8: Management of pending requests with accept().

2.4 UDP connection

UDP connection is defined by type **SOCK_DGRAM** as specified in Section 2.2. 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.

Chapter 3

HTTP protocol

HTTP protocol was presented for the first time in the RFC 1945 (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.

3.1 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.

3.2 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)>

3.3 Messages

3.3.1 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]
```

3.3.2 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
```

3.3.3 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.

3.3.4 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.

3.3.5 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
```

3.3.6 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

3.4 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.

3.4.1 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)

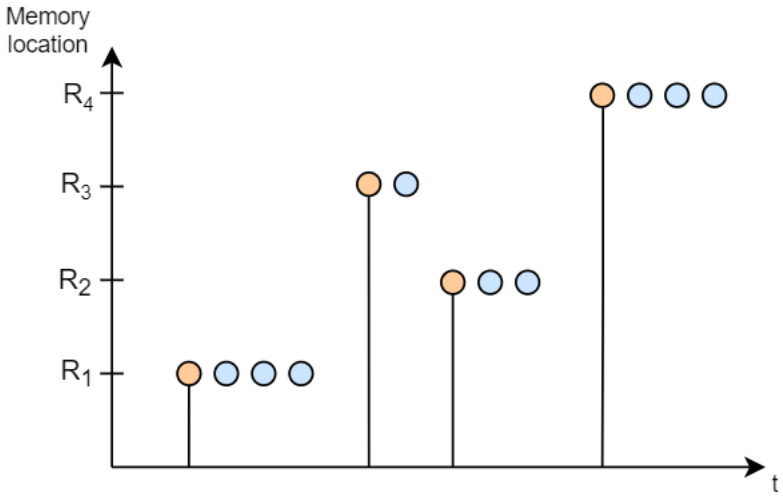
- **Location**
  defines the exact location of the resource that was identified by the Request-URI (useful for 3xx responses)
- **Pragma**
  It's sent by server to inform that there in no caching systems
- **Referer**
  allows the client to specify, for the server's benefit, the address (URI) of the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

3.4.2 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:

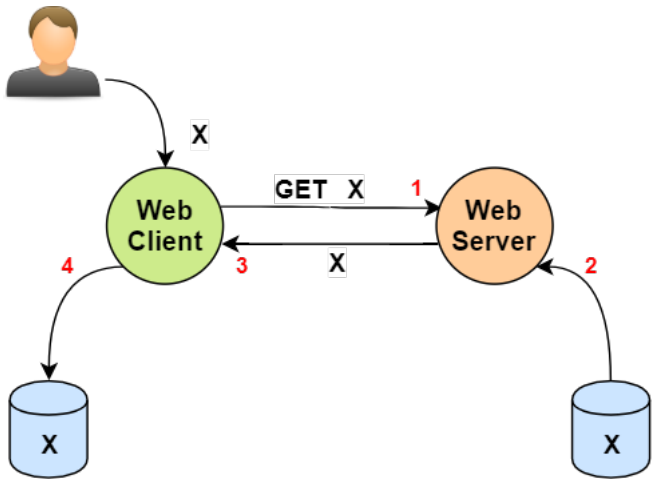
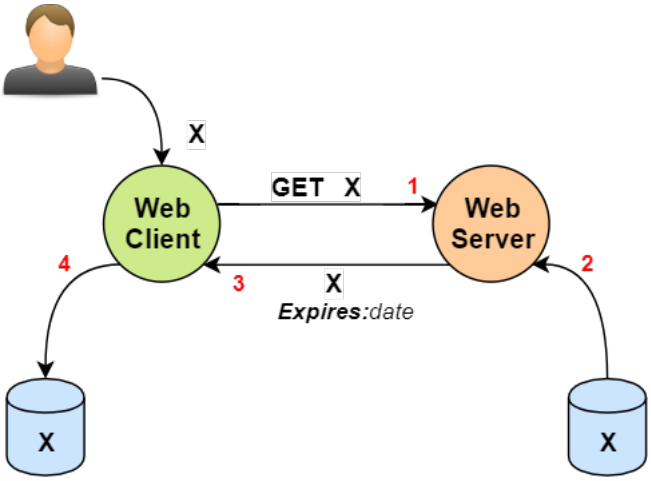


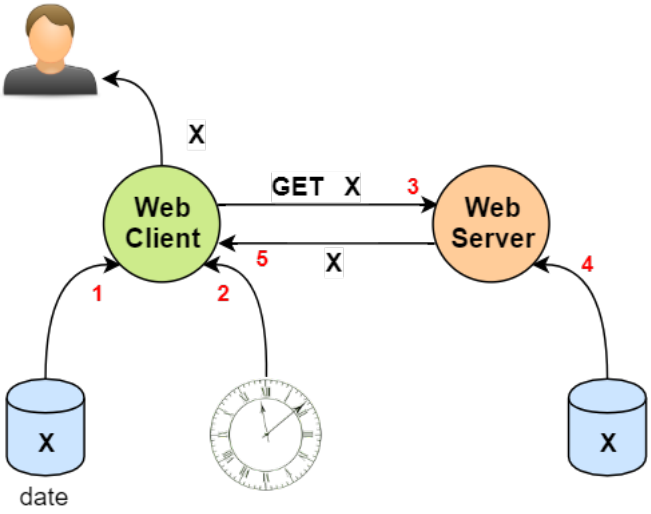
Figure 3.1: First insertion of the resource in the cache.

- **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.

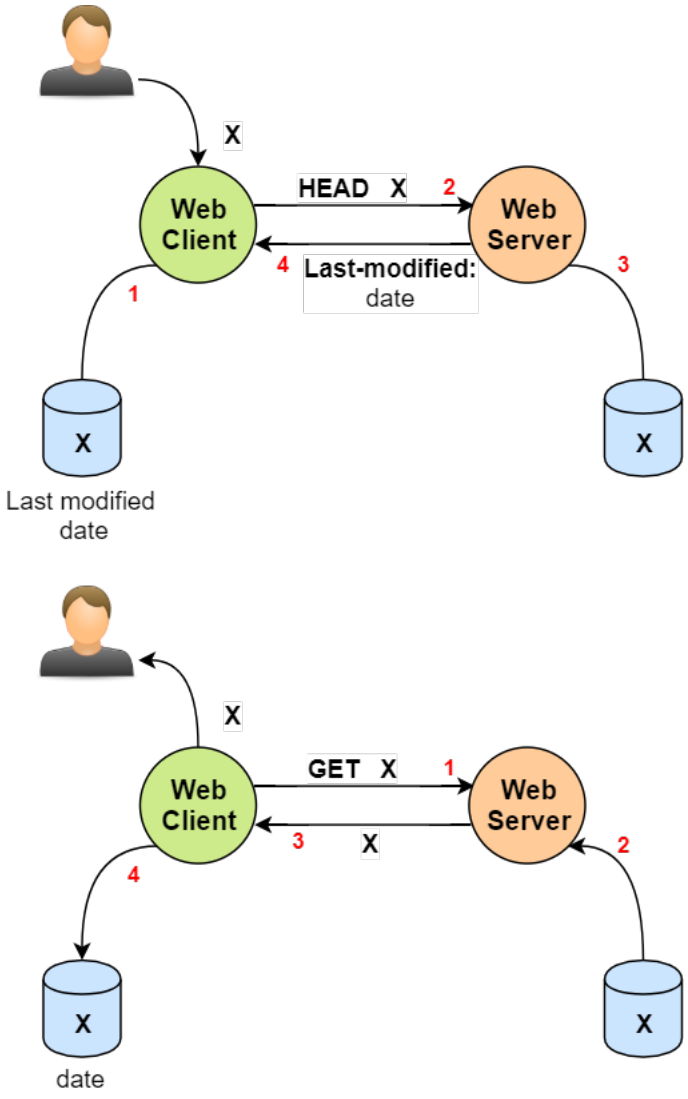

X
Web Client
GET X
1
Web Server
4
3
X
Expires:date
2
X
X


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.

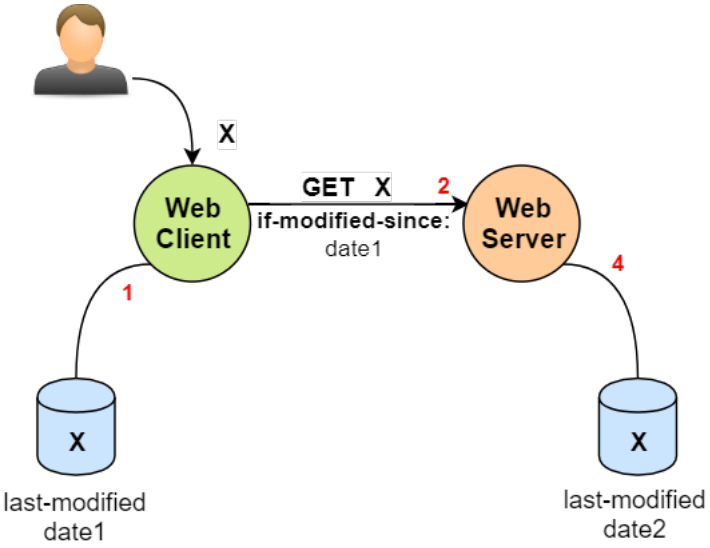
  The problem of this method is that, in the worst case, we send two times the request of the same resource (even if the first one, with "HEAD" method, is less heavy).

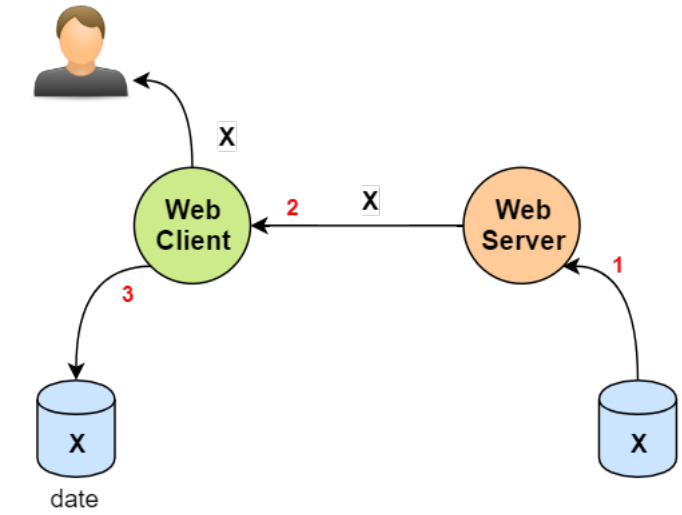


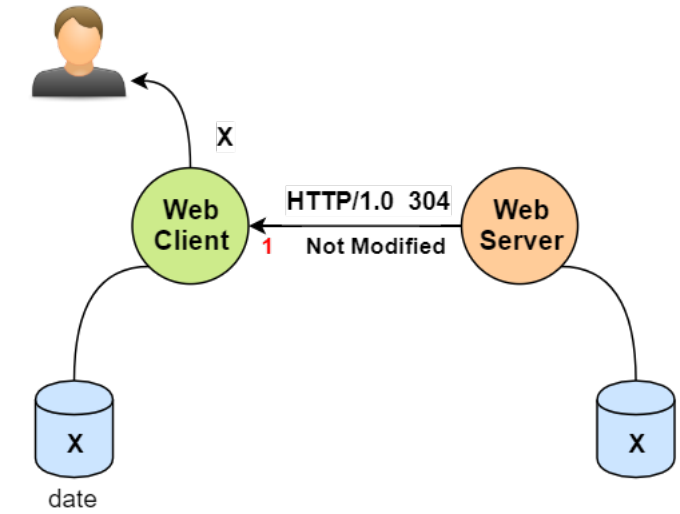
- **Request with if-modified-since header**

  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 correpct 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

3.4.3 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:

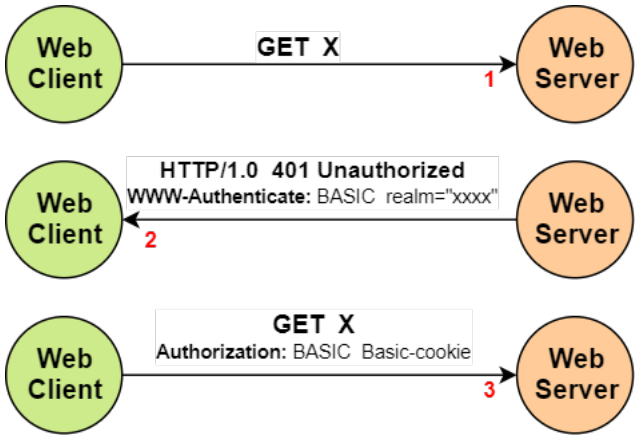
```
WWW-Authenticate: Auth-Scheme Realm="XXXX"
```

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:

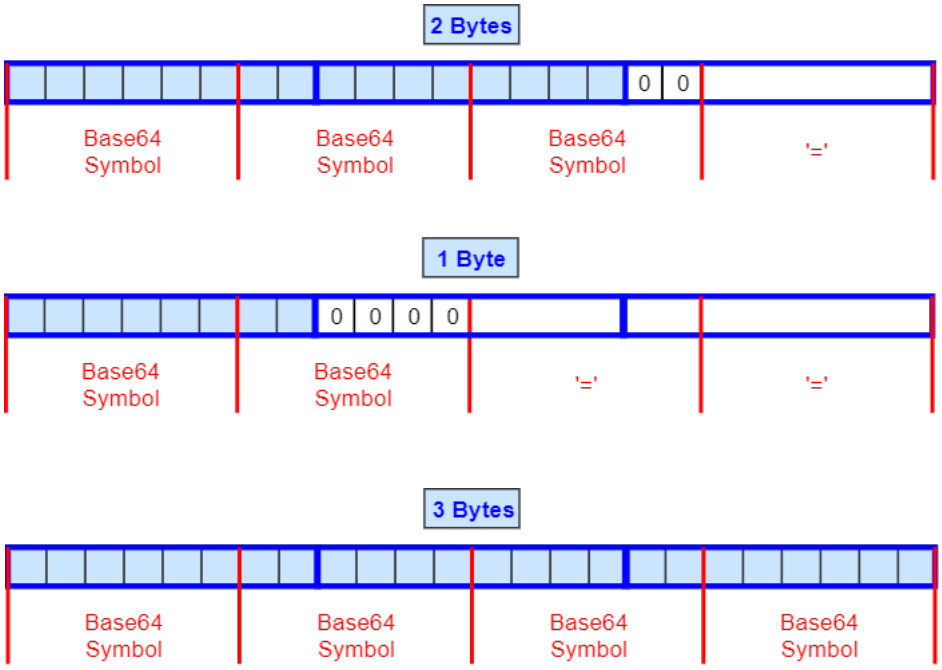
```
WWW-Authenticate: Auth-Scheme Basic-cookie
```

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)



3.4.3.1 base64

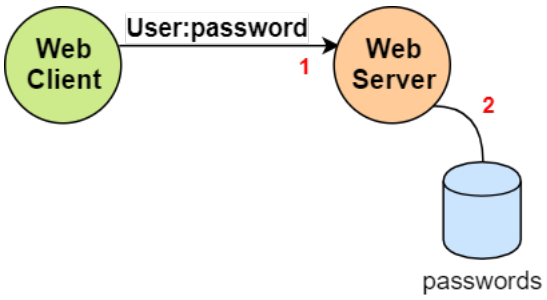
It is very useful for a lot of protocol like HTTP, that doesn't support format different than text of characters. For example with SMTP, all the mail contents must be text, hence images or other binary files are encrypted with base64.
Starting from a stream of bytes, we are going to convert numbers,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



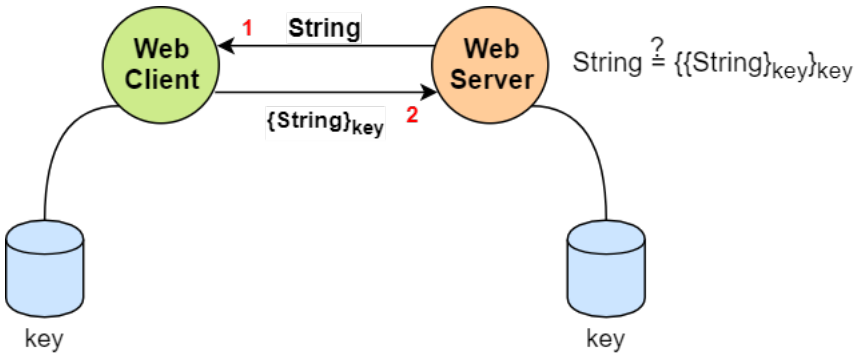
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47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

3.4.3.2 Auth-schemes

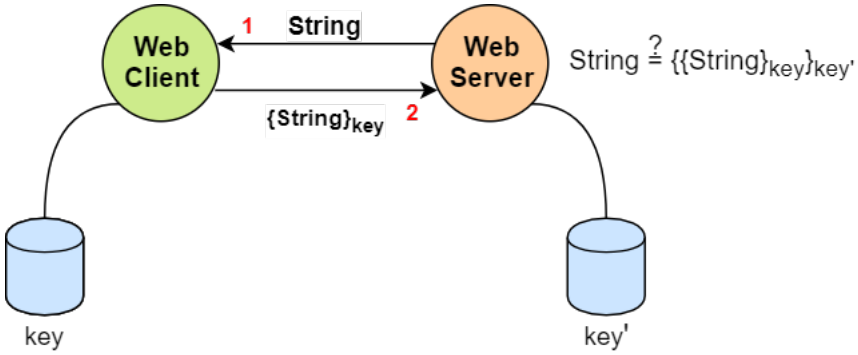
- **BASIC**



- **Challenge** (symmetric version)



- **Challenge** (asymmetric version)



3.5 HTTP 1.1

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 3.2). 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)
```

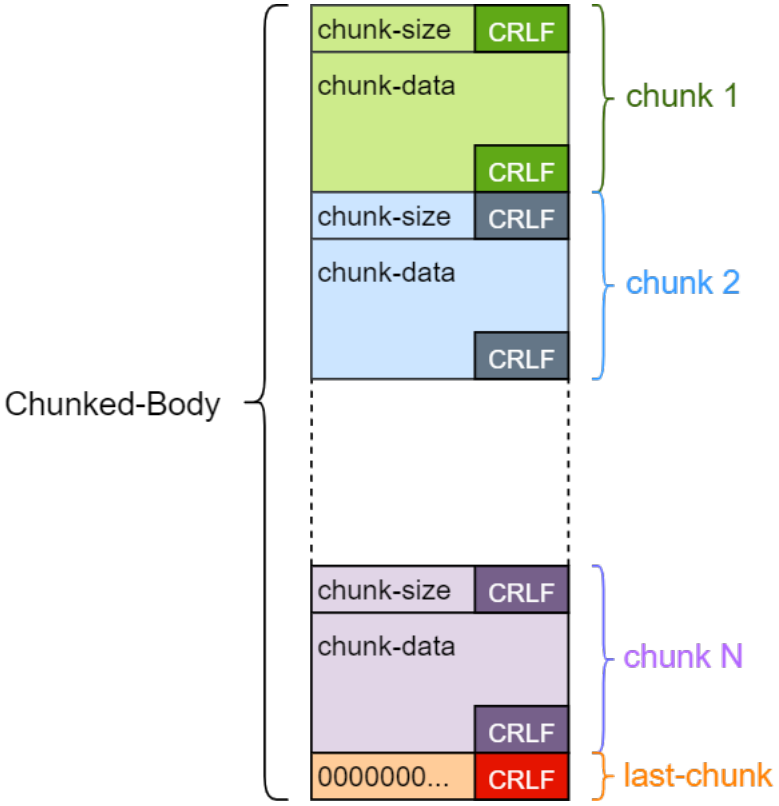


Figure 3.2: Chunked body.

3.5.1 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.

3.6 URI

In URI, there is the encapsulation of the operation done in the past, to have a resource from a server:

- Open the application ftp

- Open the server File System, through a general login

- Select the resource you want to use and download it

```
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)


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,
```

Uniform Resource Identifiers are simply formatted strings which identify–via name, location, or any other characteristic–a network resource.

```
Relative URI
//net_loc/a/b/c?query
 server    path  params
```

3.6.1 HTTP URL

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

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

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

port            = *DIGIT
```

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

3.7 HTML

The body of an HTTP request, it's often composed from the HTML related page. Each click, of a link inside the web page, generates a new request to the server with GET method.