The application Layer

The Application Layer is the most important and most visible layer in computer networks. Applications reside in this layer and human users interact via those applications through the network.

In this chapter, we first briefly describe the main principles of the application layer and focus on the two most important application models : the client-server and the peer-to-peer models. Then, we review in detail two families of protocols that have proved to be very useful in the Internet : electronic mail and the protocols that allow access to information on the world wide web. We also describe the Domain Name System that allows humans to use user-friendly names while the hosts use 32 bits or 128 bits long IP addresses.

3.1 Principles

The are two important models used to organise a networked application. The first and oldest model is the client-server model. In this model, a server provides services to clients that exchange information with it. This model is highly asymmetrical : clients send requests and servers perform actions and return responses. It is illustrated in the figure below.

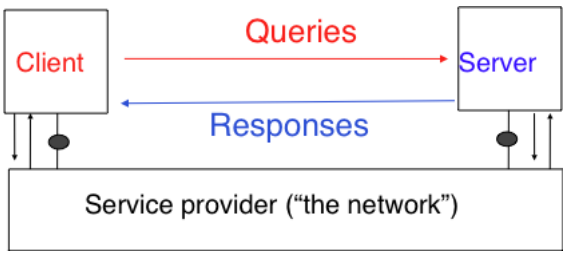

Figure 3.1: The client-server model

The client-server model was the first model to be used to develop networked applications. This model comes naturally from the mainframes and minicomputers that were the only networked computers used until the 1980s. A minicomputer is a multi-user system that is used by tens or more users at the same time. Each user interacts with the minicomputer by using a terminal. Those terminals, were mainly a screen, a keyboard and a cable directly connected to the minicomputer.

There are various types of servers as well as various types of clients. A web server provides information in response to the query sent by its clients. A print server prints documents sent as queries by the client. An email server will forward towards their recipient the email messages sent as queries while a music server will deliver the music requested by the client. From the viewpoint of the application developer, the client and the server applications directly exchange messages (the horizontal arrows labelled *Queries* and *Responses* in the above figure), but in practice these messages are exchanged thanks to the underlying layers (the vertical arrows in the above figure). In this chapter, we focus on these horizontal exchanges of messages.

Networked applications do not exchange random messages. In order to ensure that the server is able to understand the queries sent by a client, and also that the client is able to understand the responses sent by the server, they must both agree on a set of syntactical and semantic rules. These rules define the format of the messages exchanged as well as their ordering. This set of rules is called an application-level *protocol*.

An *application-level protocol* is similar to a structured conversation between humans. Assume that Alice wants to know the current time but does not have a watch. If Bob passes close by, the following conversation could take place :

- Alice : *Hello*
- Bob : *Hello*
- Alice : *What time is it ?*
- Bob : *11:55*
- Alice : *Thank you*
- Bob : *You're welcome*

Such a conversation succeeds if both Alice and Bob speak the same language. If Alice meets Tchang who only speaks Chinese, she won't be able to ask him the current time. A conversation between humans can be more complex. For example, assume that Bob is a security guard whose duty is to only allow trusted secret agents to enter a meeting room. If all agents know a secret password, the conversation between Bob and Trudy could be as follows :

- Bob : *What is the secret password ?*
- Trudy : *1234*
- Bob : *This is the correct password, you're welcome*

If Alice wants to enter the meeting room but does not know the password, her conversation could be as follows :

- Bob : *What is the secret password ?*
- Alice : *3.1415*
- Bob : *This is not the correct password.*

Human conversations can be very formal, e.g. when soldiers communicate with their hierarchy, or informal such as when friends discuss. Computers that communicate are more akin to soldiers and require well-defined rules to ensure an successful exchange of information. There are two types of rules that define how information can be exchanged between computers :

- syntactical rules that precisely define the format of the messages that are exchanged. As computers only process bits, the syntactical rules specify how information is encoded as bit strings
- organisation of the information flow. For many applications, the flow of information must be structured and there are precedence relationships between the different types of information. In the time example above, Alice must greet Bob before asking for the current time. Alice would not ask for the current time first and greet Bob afterwards. Such precedence relationships exist in networked applications as well. For example, a server must receive a username and a valid password before accepting more complex commands from its clients.

Let us first discuss the syntactical rules. We will later explain how the information flow can be organised by analysing real networked applications.

Application-layer protocols exchange two types of messages. Some protocols such as those used to support electronic mail exchange messages expressed as strings or lines of characters. As the transport layer allows hosts to exchange bytes, they need to agree on a common representation of the characters. The first and simplest method to encode characters is to use the *ASCII* table. **RFC 20** provides the ASCII table that is used by many protocols on the Internet. For example, the table defines the following binary representations :

- *A* : *1000011b*
- *0* : *0110000b*
- *z* : *1111010b*

- @ : *1000000b*
- *space* : *0100000b*

In addition, the *ASCII* table also defines several non-printable or control characters. These characters were designed to allow an application to control a printer or a terminal. These control characters include *CR* and *LF*, that are used to terminate a line, and the *Bell* character which causes the terminal to emit a sound.

- *carriage return* (*CR*) : *0001101b*
- *line feed* (*LF*) : *0001010b*
- *Bell*: *0000111b*

The *ASCII* characters are encoded as a seven bits field, but transmitted as an eight-bits byte whose high order bit is usually set to *0*. Bytes are always transmitted starting from the high order or most significant bit.

Most applications exchange strings that are composed of fixed or variable numbers of characters. A common solution to define the character strings that are acceptable is to define them as a grammar using a Backus-Naur Form (*BNF*) such as the Augmented BNF defined in **RFC 5234**. A BNF is a set of production rules that generate all valid character strings. For example, consider a networked application that uses two commands, where the user can supply a username and a password. The BNF for this application could be defined as shown in the figure below.

```
command         = usercommand / passworcommand
usercommand     = "user" SP username CRLF
passwordcommand = "pass" SP password CRLF
username        = 1*8ALPHA
password        = (ALPHA) *(ALPHA/DIGIT)
ALPHA           =  %x41-5A / %x61-7A
CR              =  %x0D
CRLF            =  CR LF
DIGIT           =  "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9"
LF              =  %x0A
SP              =  %x20 / %x09
```

Figure 3.2: A simple BNF specification

The example above defines several terminals and two commands : *usercommand* and *passwordcommand*. The *ALPHA* terminal contains all letters in upper and lower case. In the *ALPHA* rule, *%x41* corresponds to ASCII character code 41 in hexadecimal, i.e. capital *A*. The *CR* and *LF* terminals correspond to the carriage return and linefeed control characters. The *CRLF* rule concatenates these two terminals to match the standard end of line termination. The *DIGIT* terminal contains all digits. The *SP* terminal corresponds to the white space characters. The *usercommand* is composed of two strings separated by white space. In the ABNF rules that define the messages used by Internet applications, the commands are case-insensitive. The rule *"user"* corresponds to all possible cases of the letters that compose the word between brackets, e.g. *user*, *uSeR*, *USER*, *usER*, ... A *username* contains at least one letter and up to 8 letters. User names are case-sensitive as they are not defined as a string between brackets. The *password* rule indicates that a password starts with a letter and can contain any number of letters or digits. The white space and the control characters cannot appear in a *password* defined by the above rule.

Besides character strings, some applications also need to exchange 16 bits and 32 bits fields such as integers. A naive solution would have been to send the 16- or 32-bits field as it is encoded in the host's memory. Unfortunately, there are different methods to store 16- or 32-bits fields in memory. Some CPUs store the most significant byte of a 16-bits field in the first address of the field while others store the least significant byte at this location. When networked applications running on different CPUs exchange 16 bits fields, there are two possibilities to transfer them over the transport service :

- send the most significant byte followed by the least significant byte
- send the least significant byte followed by the most significant byte

The first possibility was named *big-endian* in a note written by Cohen [Cohen1980] while the second was named *little-endian*. Vendors of CPUs that used *big-endian* in memory insisted on using *big-endian* encoding in networked applications while vendors of CPUs that used *little-endian* recommended the opposite. Several studies were written on the relative merits of each type of encoding, but the discussion became almost a religious issue [Cohen1980]. Eventually, the Internet chose the *big-endian* encoding, i.e. multi-byte fields are always transmitted by sending the most significant byte first, **RFC 791** refers to this encoding as the *network-byte order*. Most

libraries [1] used to write networked applications contain functions to convert multi-byte fields from memory to the network byte order and vice versa.

Besides 16 and 32 bit words, some applications need to exchange data structures containing bit fields of various lengths. For example, a message may be composed of a 16 bits field followed by eight, one bit flags, a 24 bits field and two 8 bits bytes. Internet protocol specifications will define such a message by using a representation such as the one below. In this representation, each line corresponds to 32 bits and the vertical lines are used to delineate fields. The numbers above the lines indicate the bit positions in the 32-bits word, with the high order bit at position *0*.

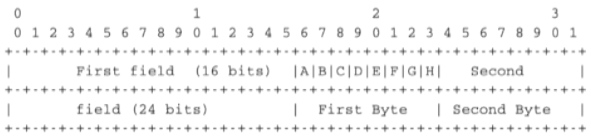

Figure 3.3: Message format

The message mentioned above will be transmitted starting from the upper 32-bits word in network byte order. The first field is encoded in 16 bits. It is followed by eight one bit flags (*A*-*H*), a 24 bits field whose high order byte is shown in the first line and the two low order bytes appear in the second line followed by two one byte fields. This ASCII representation is frequently used when defining binary protocols. We will use it for all the binary protocols that are discussed in this book.

We will discuss several examples of application-level protocols in this chapter.

3.1.1 The peer-to-peer model

The peer-to-peer model emerged during the last ten years as another possible architecture for networked applications. In the traditional client-server model, hosts act either as servers or as clients and a server serves a large number of clients. In the peer-to-peer model, all hosts act as both servers and clients and they play both roles. The peer-to-peer model has been used to develop various networked applications, ranging from Internet telephony to file sharing or Internet-wide filesystems. A detailed description of peer-to-peer applications may be found in [BYL2008]. Surveys of peer-to-peer protocols and applications may be found in [AS2004] and [LCP2005].

3.1.2 The transport services

Networked applications are built on top of the transport service. As explained in the previous chapter, there are two main types of transport services :

- the *connectionless* or *datagram* service
- the *connection-oriented* or *byte-stream* service

The connectionless service allows applications to easily exchange messages or Service Data Units. On the Internet, this service is provided by the UDP protocol that will be explained in the next chapter. The connectionless transport service on the Internet is unreliable, but is able to detect transmission errors. This implies that an application will not receive an SDU that has been corrupted due to transmission errors.

The connectionless transport service allows networked application to exchange messages. Several networked applications may be running at the same time on a single host. Each of these applications must be able to exchange SDUs with remote applications. To enable these exchanges of SDUs, each networked application running on a host is identified by the following information :

- the *host* on which the application is running
- the *port number* on which the application *listens* for SDUs

[1] For example, the `htonl(3)` (resp. `ntohl(3)`) function the standard C library converts a 32-bits unsigned integer from the byte order used by the CPU to the network byte order (resp. from the network byte order to the CPU byte order). Similar functions exist in other programming languages.

On the Internet, the *port number* is an integer and the *host* is identified by its network address. As we will see in chapter *The network layer* there are two types of Internet Addresses :

- *IP version 4* addresses that are 32 bits wide
- *IP version 6* addresses that are 128 bits wide

IPv4 addresses are usually represented by using a dotted decimal representation where each decimal number corresponds to one byte of the address, e.g. *203.0.113.56*. IPv6 addresses are usually represented as a set of hexadecimal numbers separated by semicolons, e.g. *2001:db8:3080:2:217:f2ff:fed6:65c0*. Today, most Internet hosts have one IPv4 address. A small fraction of them also have an IPv6 address. In the future, we can expect that more and more hosts will have IPv6 addresses and that some of them will not have an IPv4 address anymore. A host that only has an IPv4 address cannot communicate with a host having only an IPv6 address. The figure below illustrates two that are using the datagram service provided by UDP on hosts that are using IPv4 addresses.

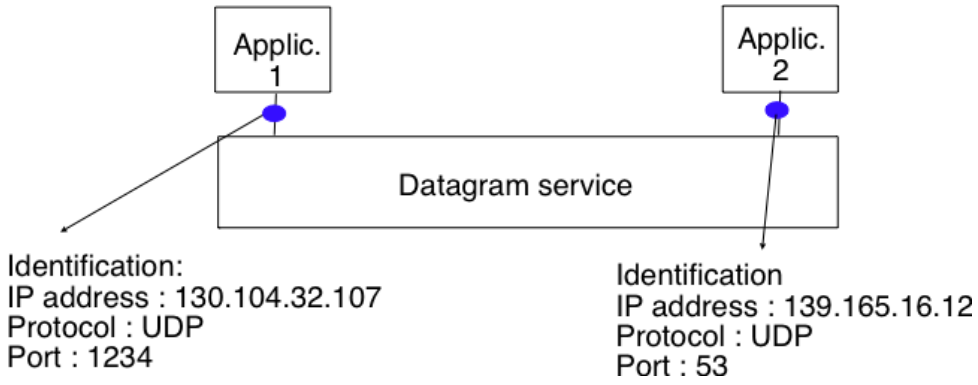

Figure 3.4: The connectionless or datagram service

The second transport service is the connection-oriented service. On the Internet, this service is often called the *byte-stream service* as it creates a reliable byte stream between the two applications that are linked by a transport connection. Like the datagram service, the networked applications that use the byte-stream service are identified by the host on which they run and a port number. These hosts can be identified by an IPv4 address, an IPv6 address or a name. The figure below illustrates two applications that are using the byte-stream service provided by the TCP protocol on IPv6 hosts. The byte stream service provided by TCP is reliable and bidirectional.

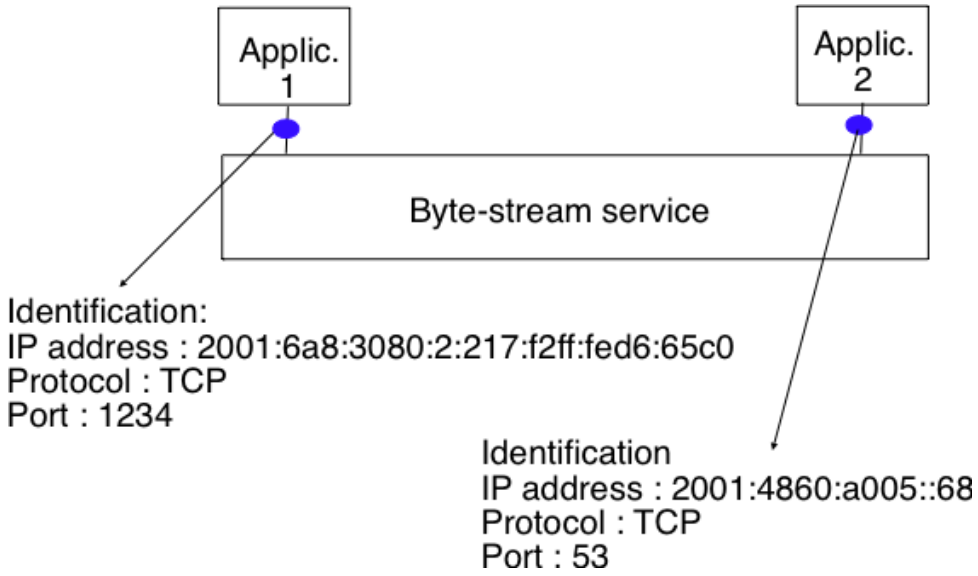

Figure 3.5: The connection-oriented or byte-stream service

3.2 Application-level protocols

Many protocols have been defined for networked applications. In this section, we describe some of the important applications that are used on the Internet. We first explain the Domain Name System (DNS) that enables hosts to be identified by human-friendly names instead of the IPv4 or IPv6 addresses that are used by the network. Then, we describe the operation of electronic mail, one of the first killer applications on the global Internet, and the protocols used on world wide web.

3.2.1 The Domain Name System

In the early days of the Internet, there were only a few number of hosts (mainly minicomputers) connected to the network. The most popular applications were remote login and file transfer. By 1983, there were already five hundred hosts attached to the Internet. Each of these hosts were identified by a unique IPv4 address. Forcing human users to remember the IPv4 addresses of the remote hosts that they want to use was not user-friendly. Human users prefer to remember names, and use them when needed. Using names as aliases for addresses is a common technique in Computer Science. It simplifies the development of applications and allows the developer to ignore the low level details. For example, by using a programming language instead of writing machine code, a developer can write software without knowing whether the variables that it uses are stored in memory or inside registers.

Because names are at a higher level than addresses, they allow (both in the example of programming above, and on the Internet) to treat addresses as mere technical identifiers, which can change at will. Only the names are stable. On today's Internet, where switching to another ISP means changing your IP addresses, the user-friendliness of domain names is less important (they are not often typed by users) but their stability remains a very important, may be their most important property.

The first solution that allowed applications to use names was the *hosts.txt* file. This file is similar to the symbol table found in compiled code. It contains the mapping between the name of each Internet host and its associated IP address [2]. It was maintained by SRI International that coordinated the Network Information Center (NIC). When a new host was connected to the network, the system administrator had to register its name and IP address at the NIC. The NIC updated the *hosts.txt* file on its server. All Internet hosts regularly retrieved the updated *hosts.txt* file from the server maintained by SRI. This file was stored at a well-known location on each Internet host (see **RFC 952**) and networked applications could use it to find the IP address corresponding to a name.

A *hosts.txt* file can be used when there are up to a few hundred hosts on the network. However, it is clearly not suitable for a network containing thousands or millions of hosts. A key issue in a large network is to define a suitable naming scheme. The ARPANet initially used a flat naming space, i.e. each host was assigned a unique name. To limit collisions between names, these names usually contained the name of the institution and a suffix to identify the host inside the institution (a kind of poor man's hierarchical naming scheme). On the ARPANet few institutions had several hosts connected to the network.

However, the limitations of a flat naming scheme became clear before the end of the ARPANet and **RFC 819** proposed a hierarchical naming scheme. While **RFC 819** discussed the possibility of organising the names as a directed graph, the Internet opted eventually for a tree structure capable of containing all names. In this tree, the top-level domains are those that are directly attached to the root. The first top-level domain was *.arpa* [3]. This top-level name was initially added as a suffix to the names of the hosts attached to the ARPANet and listed in the *hosts.txt* file. In 1984, the *.gov*, *.edu*, *.com*, *.mil* and *.org* generic top-level domain names were added and **RFC 1032** proposed the utilisation of the two letter *ISO-3166* country codes as top-level domain names. Since *ISO-3166* defines a two letter code for each country recognised by the United Nations, this allowed all countries to automatically have a top-level domain. These domains include *.be* for Belgium, *.fr* for France, *.us* for the USA, *.ie* for Ireland or *.tv* for Tuvalu, a group of small islands in the Pacific and *.tm* for Turkmenistan. Today, the set of top-level domain-names is managed by the Internet Corporation for Assigned Names and Numbers (*ICANN*). Recently, *ICANN* added a dozen of generic top-level domains that are not related to a country and the *.cat* top-level domain has been registered for the Catalan language. There are ongoing discussions within *ICANN* to increase the number of top-level domains.

[2] The *hosts.txt* file is not maintained anymore. A historical snapshot retrieved on April 15th, 1984 is available from http://ftp.univie.ac.at/netinfo/netinfo/hosts.txt

[3] See http://www.donelan.com/dnstimeline.html for a time line of DNS related developments.

Each top-level domain is managed by an organisation that decides how sub-domain names can be registered. Most top-level domain names use a first-come first served system, and allow anyone to register domain names, but there are some exceptions. For example, *.gov* is reserved for the US government, *.int* is reserved for international organisations and names in the *.ca* are mainly reserved for companies or users who are present in Canada.

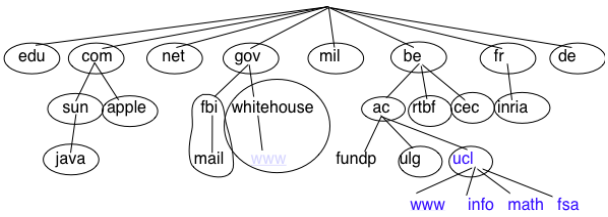

Figure 3.6: The tree of domain names

RFC 1035 recommended the following *BNF* for fully qualified domain names, to allow host names with a syntax which works with all applications (the domain names themselves have a much richer syntax).

```
domain ::= subdomain | " "
subdomain ::= label | subdomain "." label
label ::= letter [ [ ldh-str ] let-dig ]
ldh-str ::= let-dig-hyp | let-dig-hyp ldh-str
let-dig-hyp ::= let-dig | "-"
let-dig ::= letter | digit
letter ::= any one of the 52 alphabetic characters A through Z in upper case and a through z in lower case
digit ::= any one of the ten digits 0 through 9
```

Figure 3.7: BNF of the fully qualified host names

This grammar specifies that a host name is an ordered list of labels separated by the dot (.) character. Each label can contain letters, numbers and the hyphen character (-) [4]. Fully qualified domain names are read from left to right. The first label is a hostname or a domain name followed by the hierarchy of domains and ending with the root implicitly at the right. The top-level domain name must be one of the registered TLDs [5]. For example, in the above figure, *www.whitehouse.gov* corresponds to a host named *www* inside the *whitehouse* domain that belongs to the *gov* top-level domain. *info.ucl.ac.be* corresponds to the *info* domain inside the *ucl* domain that is included in the *ac* sub-domain of the *be* top-level domain.

This hierarchical naming scheme is a key component of the Domain Name System (DNS). The DNS is a distributed database that contains mappings between fully qualified domain names and IP addresses. The DNS uses the client-server model. The clients are hosts that need to retrieve the mapping for a given name. Each *nameserver* stores part of the distributed database and answers the queries sent by clients. There is at least one *nameserver* that is responsible for each domain. In the figure below, domains are represented by circles and there are three hosts inside domain *dom* (*h1*, *h2* and *h3*) and three hosts inside domain *a.sdom1.dom*. As shown in the figure below, a sub-domain may contain both host names and sub-domains.

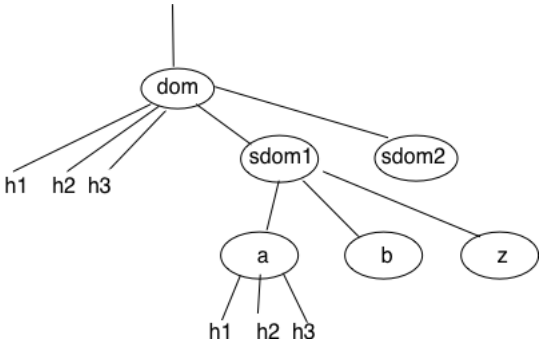

Figure 3.8: A simple tree of domain names

[4] This specification evolved later to support domain names written by using other character sets than us-ASCII **RFC 5890**. This extension is important to support languages other than English, but a detailed discussion is outside the scope of this document.

[5] The official list of top-level domain names is maintained by :term:'IANA at http://data.iana.org/TLD/tlds-alpha-by-domain.txt Additional information about these domains may be found at http://en.wikipedia.org/wiki/List_of_Internet_top-level_domains

A *nameserver* that is responsible for domain *dom* can directly answer the following queries :

- the IP address of any host residing directly inside domain *dom* (e.g. *h2.dom* in the figure above)
- the nameserver(s) that are responsible for any direct sub-domain of domain *dom* (i.e. *sdom1.dom* and *sdom2.dom* in the figure above, but not *z.sdom1.dom*)

To retrieve the mapping for host *h2.dom*, a client sends its query to the name server that is responsible for domain *.dom*. The name server directly answers the query. To retrieve a mapping for *h3.a.sdom1.dom* a DNS client first sends a query to the name server that is responsible for the *.dom* domain. This nameserver returns the nameserver that is responsible for the *sdom1.dom* domain. This nameserver can now be contacted to obtain the nameserver that is responsible for the *a.sdom1.dom* domain. This nameserver can be contacted to retrieve the mapping for the *h3.a.sdom1.dom* name. Thanks to this organisation of the nameservers, it is possible for a DNS client to obtain the mapping of any host inside the *.dom* domain or any of its subdomains. To ensure that any DNS client will be able to resolve any fully qualified domain name, there are special nameservers that are responsible for the root of the domain name hierarchy. These nameservers are called *root nameserver*. There are currently about a dozen root nameservers [6].

Each root nameserver maintains the list [7] of all the nameservers that are responsible for each of the top-level domain names and their IP addresses [8]. All root nameservers are synchronised and provide the same answers. By querying any of the root nameservers, a DNS client can obtain the nameserver that is responsible for any top-level-domain name. From this nameserver, it is possible to resolve any domain name.

To be able to contact the root nameservers, each DNS client must know their IP addresses. This implies, that DNS clients must maintain an up-to-date list of the IP addresses of the root nameservers [9]. Without this list, it is impossible to contact the root nameservers. Forcing all Internet hosts to maintain the most recent version of this list would be difficult from an operational point of view. To solve this problem, the designers of the DNS introduced a special type of DNS server : the DNS resolvers. A *resolver* is a server that provides the name resolution service for a set of clients. A network usually contains a few resolvers. Each host in these networks is configured to send all its DNS queries via one of its local resolvers. These queries are called *recursive queries* as the *resolver* must recurse through the hierarchy of nameservers to obtain the *answer*.

DNS resolvers have several advantages over letting each Internet host query directly nameservers. Firstly, regular Internet hosts do not need to maintain the up-to-date list of the IP addresses of the root servers. Secondly, regular Internet hosts do not need to send queries to nameservers all over the Internet. Furthermore, as a DNS resolver serves a large number of hosts, it can cache the received answers. This allows the resolver to quickly return answers for popular DNS queries and reduces the load on all DNS servers [JSBM2002].

The last component of the Domain Name System is the DNS protocol. The DNS protocol runs above both the datagram service and the bytestream services. In practice, the datagram service is used when short queries and responses are exchanged, and the bytestream service is used when longer responses are expected. In this section, we will only discuss the utilisation of the DNS protocol above the datagram service. This is the most frequent utilisation of the DNS.

DNS messages are composed of five parts that are named sections in **RFC 1035**. The first three sections are mandatory and the last two sections are optional. The first section of a DNS message is its *Header*. It contains information about the type of message and the content of the other sections. The second section contains the *Question* sent to the name server or resolver. The third section contains the *Answer* to the *Question*. When a client sends a DNS query, the *Answer* section is empty. The fourth section, named *Authority*, contains information about the servers that can provide an authoritative answer if required. The last section contains additional information that is supplied by the resolver or server but was not requested in the question.

The header of DNS messages is composed of 12 bytes and its structure is shown in the figure below.

The *ID* (identifier) is a 16-bits random value chosen by the client. When a client sends a question to a DNS server, it remembers the question and its identifier. When a server returns an answer, it returns in the *ID* field the identifier

[6] There are currently 13 root servers. In practice, some of these root servers are themselves implemented as a set of distinct physical servers. See http://www.root-servers.org/ for more information about the physical location of these servers.

[7] A copy of the information maintained by each root nameserver is available at http://www.internic.net/zones/root.zone

[8] Until February 2008, the root DNS servers only had IPv4 addresses. IPv6 addresses were added to the root DNS servers slowly to avoid creating problems as discussed in http://www.icann.org/en/committees/security/sac018.pdf In 2010, several DNS root servers are still not reachable by using IPv6.

[9] The current list of the IP addresses of the root nameservers is maintained at http://www.internic.net/zones/named.root . These IP addresses are stable and root nameservers seldom change their IP addresses. DNS resolvers must however maintain an up-to-date copy of this file.

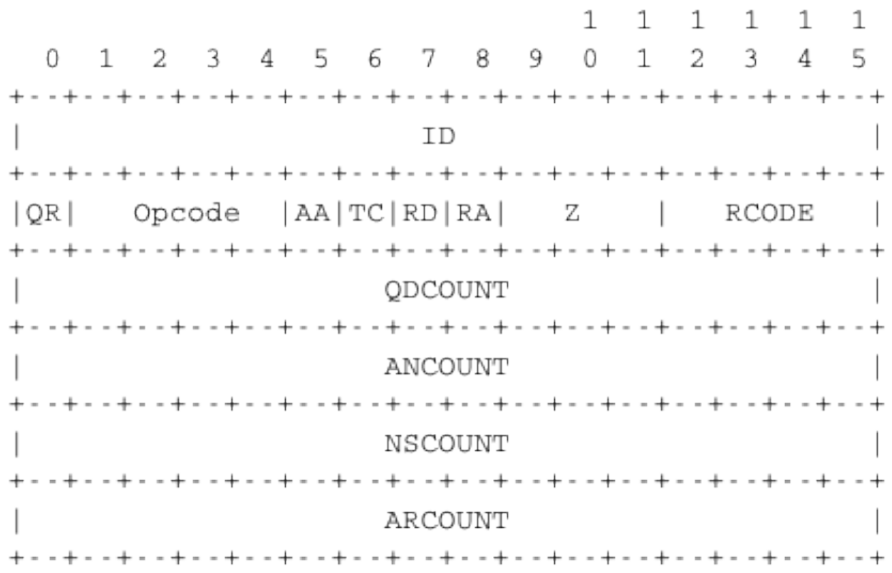

Figure 3.9: DNS header

chosen by the client. Thanks to this identifier, the client can match the received answer with the question that it sent.

The *QR* flag is set to *0* in DNS queries and *1* in DNS answers. The *Opcode* is used to specify the type of query. For instance, a *standard query* is when a client sends a *name* and the server returns the corresponding *data* and an update request is when the client sends a *name* and new *data* and the server then updates its database.

The *AA* bit is set when the server that sent the response has *authority* for the domain name found in the question section. In the original DNS deployments, two types of servers were considered : *authoritative* servers and *non-authoritative* servers. The *authoritative* servers are managed by the system administrators responsible for a given domain. They always store the most recent information about a domain. *Non-authoritative* servers are servers or resolvers that store DNS information about external domains without being managed by the owners of a domain. They may thus provide answers that are out of date. From a security point of view, the *authoritative* bit is not an absolute indication about the validity of an answer. Securing the Domain Name System is a complex problem that was only addressed satisfactorily recently by the utilisation of cryptographic signatures in the DNSSEC extensions to DNS described in **RFC 4033**. However, these extensions are outside the scope of this chapter.

The *RD* (recursion desired) bit is set by a client when it sends a query to a resolver. Such a query is said to be *recursive* because the resolver will recurse through the DNS hierarchy to retrieve the answer on behalf of the client. In the past, all resolvers were configured to perform recursive queries on behalf of any Internet host. However, this exposes the resolvers to several security risks. The simplest one is that the resolver could become overloaded by having too many recursive queries to process. As of this writing, most resolvers [10] only allow recursive queries from clients belonging to their company or network and discard all other recursive queries. The *RA* bit indicates whether the server supports recursion. The *RCODE* is used to distinguish between different types of errors. See **RFC 1035** for additional details. The last four fields indicate the size of the *Question*, *Answer*, *Authority* and *Additional* sections of the DNS message.

The last four sections of the DNS message contain *Resource Records* (RR). All RRs have the same top level format shown in the figure below.

In a *Resource Record* (*RR*), the *Name* indicates the name of the node to which this resource record pertains. The two bytes *Type* field indicate the type of resource record. The *Class* field was used to support the utilisation of the DNS in other environments than the Internet.

The *TTL* field indicates the lifetime of the *Resource Record* in seconds. This field is set by the server that returns an answer and indicates for how long a client or a resolver can store the *Resource Record* inside its cache. A long *TTL* indicates a stable *RR*. Some companies use short *TTL* values for mobile hosts and also for popular servers.

[10] Some DNS resolvers allow any host to send queries. OpenDNS and GoogleDNS are example of open resolvers.

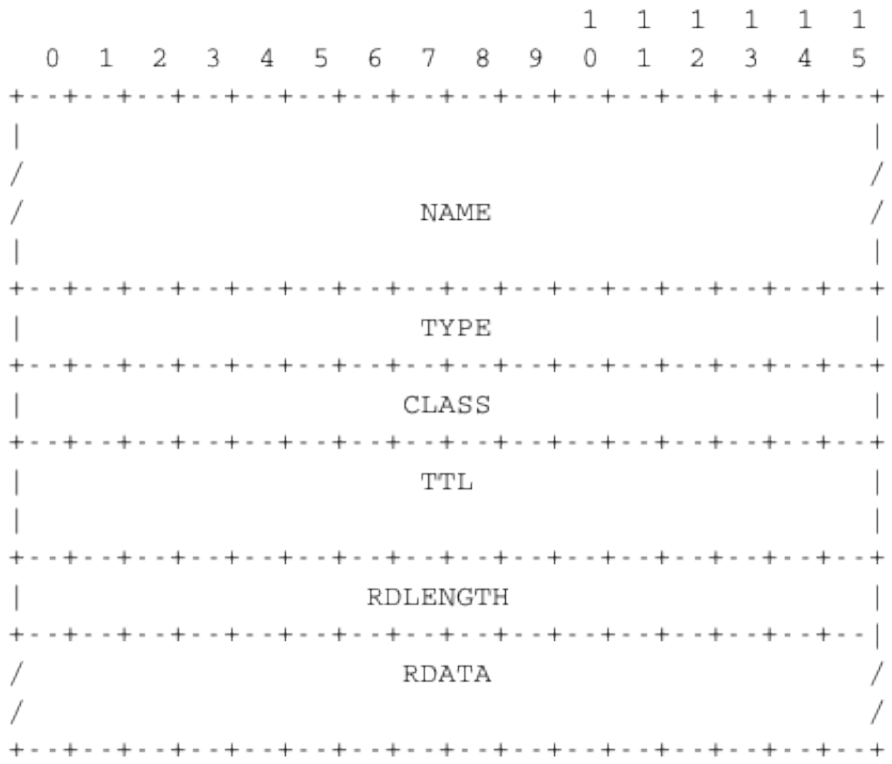

Figure 3.10: DNS Resource Records

For example, a web hosting company that wants to spread the load over a pool of hundred servers can configure its nameservers to return different answers to different clients. If each answer has a small *TTL*, the clients will be forced to send DNS queries regularly. The nameserver will reply to these queries by supplying the address of the less loaded server.

The *RDLength* field is the length of the *RData* field that contains the information of the type specified in the *Type* field.

Several types of DNS RR are used in practice. The *A* type is used to encode the IPv4 address that corresponds to the specified name. The *AAAA* type is used to encode the IPv6 address that corresponds to the specified name. A *NS* record contains the name of the DNS server that is responsible for a given domain. For example, a query for the *A* record associated to the *www.ietf.org* name returns the following answer.

This answer contains several pieces of information. First, the name *www.ietf.org* is associated to IP address *64.170.98.32*. Second, the *ietf.org* domain is managed by six different nameservers. Three of these nameservers are reachable via IPv4 and IPv6. Two of them are not reachable via IPv6 and *ns0.ietf.org* is only reachable via IPv6. A query for the *AAAA* record associated to *www.ietf.org* returns *2001:1890:1112:1::20* and the same authority and additional sections.

CNAME (or canonical names) are used to define aliases. For example *www.example.com* could be a *CNAME* for *pc12.example.com* that is the actual name of the server on which the web server for *www.example.com* runs.

Note: Reverse DNS and in-addr.arpa

The DNS is mainly used to find the IP address that correspond to a given name. However, it is sometimes useful to obtain the name that corresponds to an IP address. This done by using the *PTR* (*pointer*) *RR*. The *RData* part of a *PTR RR* contains the name while the *Name* part of the *RR* contains the IP address encoded in the *in-addr.arpa* domain. IPv4 addresses are encoded in the *in-addr.arpa* by reversing the four digits that compose the dotted decimal representation of the address. For example, consider IPv4 address *192.0.2.11*. The hostname associated to this address can be found by requesting the *PTR RR* that corresponds to *11.2.0.192.in-addr.arpa*. A similar solution is used to support IPv6 addresses, see **RFC 3596**.

```
; <<>> DiG 9.6.0-APPLE-P2 <<>> -t A www.ietf.org
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 33431
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 6, ADDITIONAL: 9
;; QUESTION SECTION:
;www.ietf.org.                  IN      A

;; ANSWER SECTION:
www.ietf.org.           1800    IN      A       64.170.98.32

;; AUTHORITY SECTION:
ietf.org.               592     IN      NS      ns0.ietf.org.
ietf.org.               592     IN      NS      ns1.yyz1.afilias-nst.info.
ietf.org.               592     IN      NS      ns1.hkg1.afilias-nst.info.
ietf.org.               592     IN      NS      ns1.ams1.afilias-nst.info.
ietf.org.               592     IN      NS      ns1.mia1.afilias-nst.info.
ietf.org.               592     IN      NS      ns1.sea1.afilias-nst.info.

;; ADDITIONAL SECTION:
ns0.ietf.org.           1800    IN      AAAA    2001:1890:1112:1::14
ns1.ams1.afilias-nst.info. 1235         IN      A       199.19.48.79
ns1.ams1.afilias-nst.info. 3204         IN      AAAA    2001:500:6::79
ns1.hkg1.afilias-nst.info. 1428         IN      A       199.19.51.79
ns1.hkg1.afilias-nst.info. 1428         IN      AAAA    2001:500:9::79
ns1.mia1.afilias-nst.info. 2870         IN      A       199.19.52.79
ns1.sea1.afilias-nst.info. 3324         IN      A       199.19.50.79
ns1.sea1.afilias-nst.info. 3314         IN      AAAA    2001:500:8::79
ns1.yyz1.afilias-nst.info. 587 IN       A       199.19.49.79
```

Figure 3.11: Query for the *A* record of *www.ietf.org*

An important point to note regarding the Domain Name System is its extensibility. Thanks to the *Type* and *RDLength* fields, the format of the Resource Records can easily be extended. Furthermore, a DNS implementation that receives a new Resource Record that it does not understand can ignore the record while still being able to process the other parts of the message. This allows, for example, a DNS server that only supports IPv4 to ignore the IPv6 addresses listed in the DNS reply for *www.ietf.org* while still being able to correctly parse the Resource Records that it understands. This extensibility allowed the Domain Name System to evolve over the years while still preserving the backward compatibility with already deployed DNS implementations.

3.2.2 Electronic mail

Electronic mail, or email, is a very popular application in computer networks such as the Internet. Email appeared in the early 1970s and allows users to exchange text based messages. Initially, it was mainly used to exchange short messages, but over the years its usage has grown. It is now not only used to exchange small, but also long messages that can be composed of several parts as we will see later.

Before looking at the details of Internet email, let us consider a simple scenario illustrated in the figure below, where Alice sends an email to Bob. Alice prepares her email by using an email clients and sends it to her email server. Alice's email server extracts Bob's address from the email and delivers the message to Bob's server. Bob retrieves Alice's message on his server and reads it by using his favourite email client or through his webmail interface.

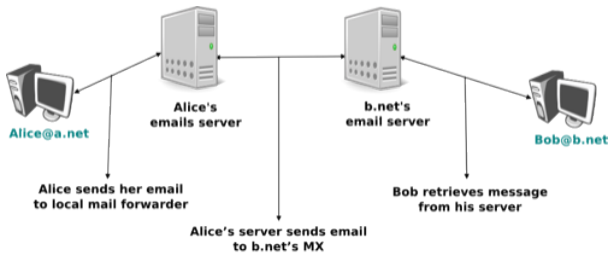

Figure 3.12: Simplified architecture of the Internet email

The email system that we consider in this book is composed of four components :

- a message format, that defines how valid email messages are encoded
- protocols, that allow hosts and servers to exchange email messages

- client software, that allows users to easily create and read email messages
- software, that allows servers to efficiently exchange email messages

We will first discuss the format of email messages followed by the protocols that are used on today's Internet to exchange and retrieve emails. Other email systems have been developed in the past [Bush1993] [Genilloud1990] [GC2000], but today most email solutions have migrated to the Internet email. Information about the software that is used to compose and deliver emails may be found on wikipedia among others, for both email clients and email servers. More detailed information about the full Internet Mail Architecture may be found in **RFC 5598**.

Email messages, like postal mail, are composed of two parts :

- a *header* that plays the same role as the letterhead in regular mail. It contains metadata about the message.
- the *body* that contains the message itself.

Email messages are entirely composed of lines of ASCII characters. Each line can contain up to 998 characters and is terminated by the *CR* and *LF* control characters **RFC 5322**. The lines that compose the *header* appear before the message *body*. An empty line, containing only the *CR* and *LF* characters, marks the end of the *header*. This is illustrated in the figure below.

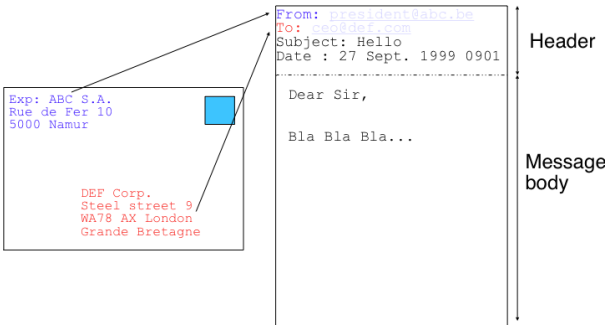

Figure 3.13: The structure of email messages

The email header contains several lines that all begin with a keyword followed by a colon and additional information. The format of email messages and the different types of header lines are defined in **RFC 5322**. Two of these header lines are mandatory and must appear in all email messages :

- The sender address. This header line starts with *From:*. This contains the (optional) name of the sender followed by its email address between < and >. Email addresses are always composed of a username followed by the @ sign and a domain name.
- The date. This header line starts with *Date:*. **RFC 5322** precisely defines the format used to encode a date.

Other header lines appear in most email messages. The *Subject:* header line allows the sender to indicate the topic discussed in the email. Three types of header lines can be used to specify the recipients of a message :

- the *To:* header line contains the email addresses of the primary recipients of the message [11] . Several addresses can be separated by using commas.
- the *cc:* header line is used by the sender to provide a list of email addresses that must receive a carbon copy of the message. Several addresses can be listed in this header line, separated by commas. All recipients of the email message receive the *To:* and *cc:* header lines.
- the *bcc:* header line is used by the sender to provide a list of comma separated email addresses that must receive a blind carbon copy of the message. The *bcc:* header line is not delivered to the recipients of the email message.

A simple email message containing the *From:*, *To:*, *Subject:* and *Date:* header lines and two lines of body is shown below.

[11] It could be surprising that the *To:* is not mandatory inside an email message. While most email messages will contain this header line an email that does not contain a *To:* header line and that relies on the *bcc:* to specify the recipient is valid as well.

```
From: Bob Smith <Bob@machine.example>
To: Alice Doe <alice@example.net>, Alice Smith <Alice@machine.example>
Subject: Hello
Date: Mon, 8 Mar 2010 19:55:06 -0600

This is the "Hello world" of email messages.
This is the second line of the body
```

Note the empty line after the *Date:* header line; this empty line contains only the *CR* and *LF* characters, and marks the boundary between the header and the body of the message.

Several other optional header lines are defined in **RFC 5322** and elsewhere [12]. Furthermore, many email clients and servers define their own header lines starting from *X-*. Several of the optional header lines defined in **RFC 5322** are worth being discussed here :

- the *Message-Id:* header line is used to associate a "unique" identifier to each email. Email identifiers are usually structured like *string@domain* where *string* is a unique character string or sequence number chosen by the sender of the email and *domain* the domain name of the sender. Since domain names are unique, a host can generate globally unique message identifiers concatenating a locally unique identifier with its domain name.
- the *In-reply-to:* is used when a message was created in reply to a previous message. In this case, the end of the *In-reply-to:* line contains the identifier of the original message.
- the *Received:* header line is used when an email message is processed by several servers before reaching its destination. Each intermediate email server adds a *Received:* header line. These header lines are useful to debug problems in delivering email messages.

The figure below shows the header lines of one email message. The message originated at a host named *wira.firstpr.com.au* and was received by *smtp3.sgsi.ucl.ac.be*. The *Received:* lines have been wrapped for readability.

```
Received: from smtp3.sgsi.ucl.ac.be (Unknown [10.1.5.3])
    by mmp.sipr-dc.ucl.ac.be
    (Sun Java(tm) System Messaging Server 7u3-15.01 64bit (built Feb 12 2010))
    with ESMTP id <0KYY00L85LI5JLE0@mmp.sipr-dc.ucl.ac.be>; Mon,
    08 Mar 2010 11:37:17 +0100 (CET)
Received: from mail.ietf.org (mail.ietf.org [64.170.98.32])
    by smtp3.sgsi.ucl.ac.be (Postfix) with ESMTP id B92351C60D7; Mon,
    08 Mar 2010 11:36:51 +0100 (CET)
Received: from [127.0.0.1] (localhost [127.0.0.1])     by core3.amsl.com (Postfix)
    with ESMTP id F066A3A68B9; Mon, 08 Mar 2010 02:36:38 -0800 (PST)
Received: from localhost (localhost [127.0.0.1])       by core3.amsl.com (Postfix)
    with ESMTP id A1E6C3A681B  for <rrg@core3.amsl.com>; Mon,
    08 Mar 2010 02:36:37 -0800 (PST)
Received: from mail.ietf.org ([64.170.98.32])
    by localhost (core3.amsl.com [127.0.0.1]) (amavisd-new, port 10024)
    with ESMTP id erw8ih2v8VQa for <rrg@core3.amsl.com>; Mon,
    08 Mar 2010 02:36:36 -0800 (PST)
Received: from gair.firstpr.com.au (gair.firstpr.com.au [150.101.162.123])
    by core3.amsl.com (Postfix) with ESMTP id 03E893A67ED      for <rrg@irtf.org>; Mon,
    08 Mar 2010 02:36:35 -0800 (PST)
Received: from [10.0.0.6] (wira.firstpr.com.au [10.0.0.6])
    by gair.firstpr.com.au (Postfix) with ESMTP id D0A49175B63; Mon,
    08 Mar 2010 21:36:37 +1100 (EST)
Date: Mon, 08 Mar 2010 21:36:38 +1100
From: Robin Whittle <rw@firstpr.com.au>
Subject: Re: [rrg] Recommendation and what happens next
In-reply-to: <C7B9C21A.4FAB%tony.li@tony.li>
To: RRG <rrg@irtf.org>
Message-id: <4B94D336.7030504@firstpr.com.au>
```

[12] The list of all standard email header lines may be found at http://www.iana.org/assignments/message-headers/message-header-index.html

```
Message content removed
```

Initially, email was used to exchange small messages of ASCII text between computer scientists. However, with the growth of the Internet, supporting only ASCII text became a severe limitation for two reasons. First of all, non-English speakers wanted to write emails in their native language that often required more characters than those of the ASCII character table. Second, many users wanted to send other content than just ASCII text by email such as binary files, images or sound.

To solve this problem, the IETF developed the Multipurpose Internet Mail Extensions (*MIME*). These extensions were carefully designed to allow Internet email to carry non-ASCII characters and binary files without breaking the email servers that were deployed at that time. This requirement for backward compatibility forced the MIME designers to develop extensions to the existing email message format **RFC 822** instead of defining a completely new format that would have been better suited to support the new types of emails.

RFC 2045 defines three new types of header lines to support MIME :

- The *MIME-Version:* header indicates the version of the MIME specification that was used to encode the email message. The current version of MIME is 1.0. Other versions of MIME may be defined in the future. Thanks to this header line, the software that processes email messages will be able to adapt to the MIME version used to encode the message. Messages that do not contain this header are supposed to be formatted according to the original **RFC 822** specification.
- The *Content-Type:* header line indicates the type of data that is carried inside the message (see below)
- The *Content-Transfer-Encoding:* header line is used to specify how the message has been encoded. When MIME was designed, some email servers were only able to process messages containing characters encoded using the 7 bits ASCII character set. MIME allows the utilisation of other character encodings.

Inside the email header, the *Content-Type:* header line indicates how the MIME email message is structured. **RFC 2046** defines the utilisation of this header line. The two most common structures for MIME messages are :

- *Content-Type: multipart/mixed*. This header line indicates that the MIME message contains several independent parts. For example, such a message may contain a part in plain text and a binary file.
- *Content-Type: multipart/alternative*. This header line indicates that the MIME message contains several representations of the same information. For example, a *multipart/alternative* message may contain both a plain text and an HTML version of the same text.

To support these two types of MIME messages, the recipient of a message must be able to extract the different parts from the message. In **RFC 822**, an empty line was used to separate the header lines from the body. Using an empty line to separate the different parts of an email body would be difficult as the body of email messages often contains one or more empty lines. Another possible option would be to define a special line, e.g. **-LAST_LINE-** to mark the boundary between two parts of a MIME message. Unfortunately, this is not possible as some emails may contain this string in their body (e.g. emails sent to students to explain the format of MIME messages). To solve this problem, the *Content-Type:* header line contains a second parameter that specifies the string that has been used by the sender of the MIME message to delineate the different parts. In practice, this string is often chosen randomly by the mail client.

The email message below, copied from **RFC 2046** shows a MIME message containing two parts that are both in plain text and encoded using the ASCII character set. The string *simple boundary* is defined in the *Content-Type:* header as the marker for the boundary between two successive parts. Another example of MIME messages may be found in **RFC 2046**.

```
Date: Mon, 20 Sep 1999 16:33:16 +0200
From: Nathaniel Borenstein <nsb@bellcore.com>
To: Ned Freed <ned@innosoft.com>
Subject: Test
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary="simple boundary"

preamble, to be ignored

--simple boundary
Content-Type: text/plain; charset=us-ascii
```

```
First part

--simple boundary
Content-Type: text/plain; charset=us-ascii

Second part
--simple boundary
```

The *Content-Type:* header can also be used inside a MIME part. In this case, it indicates the type of data placed in this part. Each data type is specified as a type followed by a subtype. A detailed description may be found in **RFC 2046**. Some of the most popular *Content-Type:* header lines are :

- *text*. The message part contains information in textual format. There are several subtypes : *text/plain* for regular ASCII text, *text/html* defined in **RFC 2854** for documents in *HTML* format or the *text/enriched* format defined in **RFC 1896**. The *Content-Type:* header line may contain a second parameter that specifies the character set used to encode the text. *charset=us-ascii* is the standard ASCII character table. Other frequent character sets include *charset=UTF8* or *charset=iso-8859-1*. The list of standard character sets is maintained by *IANA*
- *image*. The message part contains a binary representation of an image. The subtype indicates the format of the image such as gif, jpg or png.
- *audio*. The message part contains an audio clip. The subtype indicates the format of the audio clip like wav or mp3
- *video*. The message part contains a video clip. The subtype indicates the format of the video clip like avi or mp4
- *application*. The message part contains binary information that was produced by the particular application listed as the subtype. Email clients use the subtype to launch the application that is able to decode the received binary information.

Note: From ASCII to Unicode

The first computers used different techniques to represent characters in memory and on disk. During the 1960s, computers began to exchange information via tape or telephone lines. Unfortunately, each vendor had its own proprietary character set and exchanging data between computers from different vendors was often difficult. The 7 bits ASCII character table **RFC 20** set was adopted by several vendors and by many Internet protocols. However, ASCII became a problem with the internationalisation of the Internet and the desire of more and more users to use character sets that support their own written language. A first attempt at solving this problem was the definition of the ISO-8859 character sets by *ISO*. This family of standards specified various character sets that allowed the representation of many European written languages by using 8 bits characters. Unfortunately, an 8-bits character set is not sufficient to support some widely used languages, such as those used in Asian countries. Fortunately, at the end of the 1980s, several computer scientists proposed to develop a standard that supports all written languages used on Earth today. The Unicode standard [Unicode] has now been adopted by most computer and software vendors. For example, Java uses Unicode natively to manipulate characters, Python can handle both ASCII and Unicode characters. Internet applications are slowly moving towards complete support for the Unicode character sets, but moving from ASCII to Unicode is an important change that can have a huge impact on current deployed implementations. See for example, the work to completely internationalise email **RFC 4952** and domain names **RFC 5890**.

The last MIME header line is *Content-Transfer-Encoding:*. This header line is used after the *Content-Type:* header line, within a message part, and specifies how the message part has been encoded. The default encoding is to use 7 bits ASCII. The most frequent encodings are *quoted-printable* and *Base64*. Both support encoding a sequence of bytes into a set of ASCII lines that can be safely transmitted by email servers. *quoted-printable* is defined in **RFC 2045**. We briefly describe *base64* which is defined in **RFC 2045** and **RFC 4648**.

Base64 divides the sequence of bytes to be encoded into groups of three bytes (with the last group possibly being partially filled). Each group of three bytes is then divided into four six-bit fields and each six bit field is encoded as a character from the table below.

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

The example below, from **RFC 4648**, illustrates the *Base64* encoding.

Input data	0x14fb9c03d97e
8-bit	00010100 11111011 10011100 00000011 11011001 01111110
6-bit	000101 001111 101110 011100 000000 111101 100101 111110
Decimal	5 15 46 28 0 61 37 62
Encoding	F P u c A 9 l +

The last point to be discussed about *base64* is what happens when the length of the sequence of bytes to be encoded is not a multiple of three. In this case, the last group of bytes may contain one or two bytes instead of three. *Base64* reserves the = character as a padding character. This character is used twice when the last group contains two bytes and once when it contains one byte as illustrated by the two examples below.

Input data	0x14
8-bit	00010100
6-bit	000101 000000
Decimal	5 0
Encoding	F A = =

Input data	0x14b9
8-bit	00010100 11111011
6-bit	000101 001111 101100
Decimal	5 15 44
Encoding	F P s =

Now that we have explained the format of the email messages, we can discuss how these messages can be exchanged through the Internet. The figure below illustrates the protocols that are used when *Alice* sends an email message to *Bob*. *Alice* prepares her email with an email client or on a webmail interface. To send her email to *Bob*, *Alice*'s client will use the Simple Mail Transfer Protocol (*SMTP*) to deliver her message to her SMTP server. *Alice*'s email client is configured with the name of the default SMTP server for her domain. There is usually at least one SMTP server per domain. To deliver the message, *Alice*'s SMTP server must find the SMTP server that contains *Bob*'s mailbox. This can be done by using the Mail eXchange (MX) records of the DNS. A set of MX records can be associated to each domain. Each MX record contains a numerical preference and the fully qualified domain name of a SMTP server that is able to deliver email messages destined to all valid email addresses of this domain. The DNS can return several MX records for a given domain. In this case, the server with the lowest preference is used first. If this server is not reachable, the second most preferred server is used etc. *Bob*'s SMTP server will store the message sent by *Alice* until *Bob* retrieves it using a webmail interface or protocols such as the Post Office Protocol (*POP*) or the Internet Message Access Protocol (*IMAP*).

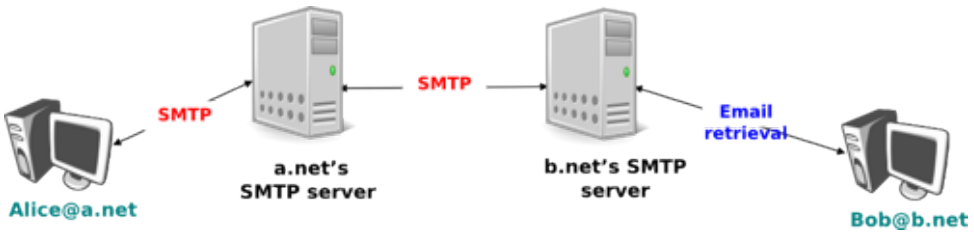

Figure 3.14: Email delivery protocols

The Simple Mail Transfer Protocol

The Simple Mail Transfer Protocol (*SMTP*) defined in **RFC 5321** is a client-server protocol. The SMTP specification distinguishes between five types of processes involved in the delivery of email messages. Email messages are composed on a Mail User Agent (MUA). The MUA is usually either an email client or a webmail. The MUA sends the email message to a Mail Submission Agent (MSA). The MSA processes the received email and forwards it to the Mail Transmission Agent (MTA). The MTA is responsible for the transmission of the email, directly or via intermediate MTAs to the MTA of the destination domain. This destination MTA will then forward the message to the Mail Delivery Agent (MDA) where it will be accessed by the recipient's MUA. SMTP is used for the interactions between MUA and MSA [13], MSA-MTA and MTA-MTA.

SMTP is a text-based protocol like many other application-layer protocols on the Internet. It relies on the byte-stream service. Servers listen on port *25*. Clients send commands that are each composed of one line of ASCII text terminated by *CR+LF*. Servers reply by sending ASCII lines that contain a three digit numerical error/success code and optional comments.

The SMTP protocol, like most text-based protocols, is specified as a *BNF*. The full BNF is defined in **RFC 5321**. The main SMTP commands are defined by the BNF rules shown in the figure below.

```
helo = "HELO" SP Domain CRLF
mail = "MAIL FROM:" Path CRLF
rcpt = "RCPT TO:" ( "<Postmaster@" Domain ">" / "<Postmaster>" / Path ) CRLF
data = "DATA" CRLF
quit = "QUIT" CRLF
Path            = "<" Mailbox ">"
Domain          = sub-domain *("." sub-domain)
sub-domain      = Let-dig [Ldh-str]
Let-dig         = ALPHA / DIGIT
Ldh-str         = *( ALPHA / DIGIT / "-" ) Let-dig
Mailbox         = Local-part "@" Domain
Local-part      = Dot-string
Dot-string      = Atom *("."  Atom)
Atom            = 1*atext
```

Figure 3.15: BNF specification of the SMTP commands

In this BNF, *atext* corresponds to printable ASCII characters. This BNF rule is defined in **RFC 5322**. The five main commands are *EHLO*, *MAIL FROM:*, *RCPT TO:*, *DATA* and *QUIT* [14]. *Postmaster* is the alias of the system administrator who is responsible for a given domain or SMTP server. All domains must have a *Postmaster* alias.

The SMTP responses are defined by the BNF shown in the figure below.

```
Greeting        = "220 " Domain [ SP textstring ] CRLF
textstring      = 1*atext
Reply-line      = *( Reply-code "-" [ textstring ] CRLF )
                  Reply-code [ SP textstring ] CRLF
Reply-code      = %x32-35 %x30-35 %x30-39
```

Figure 3.16: BNF specification of the SMTP responses

SMTP servers use structured reply codes containing three digits and an optional comment. The first digit of

[13] During the last years, many Internet Service Providers, campus and enterprise networks have deployed SMTP extensions **RFC 4954** on their MSAs. These extensions force the MUAs to be authenticated before the MSA accepts an email message from the MUA.

[14] The first versions of SMTP used *HELO* as the first command sent by a client to a SMTP server. When SMTP was extended to support newer features such as 8 bits characters, it was necessary to allow a server to recognise whether it was interacting with a client that supported the extensions or not. *EHLO* became mandatory with the publication of **RFC 2821**.

the reply code indicates whether the command was successful or not. A reply code of *2xy* indicates that the command has been accepted. A reply code of *3xy* indicates that the command has been accepted, but additional information from the client is expected. A reply code of *4xy* indicates a transient negative reply. This means that for some reason, which is indicated by either the other digits or the comment, the command cannot be processed immediately, but there is some hope that the problem will only be transient. This is basically telling the client to try the same command again later. In contrast, a reply code of *5xy* indicates a permanent failure or error. In this case, it is useless for the client to retry the same command later. Other application layer protocols such as FTP **RFC 959** or HTTP **RFC 2616** use a similar structure for their reply codes. Additional details about the other reply codes may be found in **RFC 5321**.

Examples of SMTP reply codes include the following :

```
500  Syntax error, command unrecognized
501  Syntax error in parameters or arguments
502  Command not implemented
503  Bad sequence of commands
220  <domain> Service ready
221  <domain> Service closing transmission channel
421  <domain> Service not available, closing transmission channel
250  Requested mail action okay, completed
450  Requested mail action not taken: mailbox unavailable
452  Requested action not taken: insufficient system storage
550  Requested action not taken: mailbox unavailable
354  Start mail input; end with <CRLF>.<CRLF>
```

The first four reply codes correspond to errors in the commands sent by the client. The fourth reply code would be sent by the server when the client sends commands in an incorrect order (e.g. the client tries to send an email before providing the destination address of the message). Reply code *220* is used by the server as the first message when it agrees to interact with the client. Reply code *221* is sent by the server before closing the underlying transport connection. Reply code *421* is returned when there is a problem (e.g. lack of memory/disk resources) that prevents the server from accepting the transport connection. Reply code *250* is the standard positive reply that indicates the success of the previous command. Reply codes *450* and *452* indicate that the destination mailbox is temporarily unavailable, for various reasons, while reply code *550* indicates that the mailbox does not exist or cannot be used for policy reasons. Reply code *354* indicates that the client can start transmitting its email message.

The transfer of an email message is performed in three phases. During the first phase, the client opens a transport connection with the server. Once the connection has been established, the client and the server exchange greetings messages (*EHLO* command). Most servers insist on receiving valid greeting messages and some of them drop the underlying transport connection if they do not receive a valid greeting. Once the greetings have been exchanged, the email transfer phase can start. During this phase, the client transfers one or more email messages by indicating the email address of the sender (*MAIL FROM:* command), the email address of the recipient (*RCPT TO:* command) followed by the headers and the body of the email message (*DATA* command). Once the client has finished sending all its queued email messages to the SMTP server, it terminates the SMTP association (*QUIT* command).

A successful transfer of an email message is shown below

```
S: 220 smtp.example.com ESMTP MTA information
C: EHLO mta.example.org
S: 250 Hello mta.example.org, glad to meet you
C: MAIL FROM:<alice@example.org>
S: 250 Ok
C: RCPT TO:<bob@example.com>
S: 250 Ok
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: From: "Alice Doe" <alice@example.org>
C: To: Bob Smith <bob@example.com>
C: Date: Mon, 9 Mar 2010 18:22:32 +0100
C: Subject: Hello
C:
C: Hello Bob
C: This is a small message containing 4 lines of text.
C: Best regards,
```

```
C: Alice
C: .
S: 250 Ok: queued as 12345
C: QUIT
S: 221 Bye
```

In the example above, the MTA running on *mta.example.org* opens a TCP connection to the SMTP server on host *smtp.example.com*. The lines prefixed with *S:* (resp. *C:*) are the responses sent by the server (resp. the commands sent by the client). The server sends its greetings as soon as the TCP connection has been established. The client then sends the *EHLO* command with its fully qualified domain name. The server replies with reply-code *250* and sends its greetings. The SMTP association can now be used to exchange an email.

To send an email, the client must first provide the address of the recipient with *RCPT TO:*. Then it uses the *MAIL FROM:* with the address of the sender. Both the recipient and the sender are accepted by the server. The client can now issue the *DATA* command to start the transfer of the email message. After having received the *354* reply code, the client sends the headers and the body of its email message. The client indicates the end of the message by sending a line containing only the . (dot) character [15]. The server confirms that the email message has been queued for delivery or transmission with a reply code of *250*. The client issues the *QUIT* command to close the session and the server confirms with reply-code *221*, before closing the TCP connection.

Note: Open SMTP relays and spam

Since its creation in 1971, email has been a very useful tool that is used by many users to exchange lots of information. In the early days, all SMTP servers were open and anyone could use them to forward emails towards their final destination. Unfortunately, over the years, some unscrupulous users have found ways to use email for marketing purposes or to send malware. The first documented abuse of email for marketing purposes occurred in 1978 when a marketer who worked for a computer vendor sent a marketing email to many ARPANET users. At that time, the ARPANET could only be used for research purposes and this was an abuse of the acceptable use policy. Unfortunately, given the extremely low cost of sending emails, the problem of unsolicited emails has not stopped. Unsolicited emails are now called spam and a study carried out by ENISA in 2009 reveals that 95% of email was spam and this number seems to continue to grow. This places a burden on the email infrastructure of Internet Service Providers and large companies that need to process many useless messages.

Given the amount of spam messages, SMTP servers are no longer open **RFC 5068**. Several extensions to SMTP have been developed in recent years to deal with this problem. For example, the SMTP authentication scheme defined in **RFC 4954** can be used by an SMTP server to authenticate a client. Several techniques have also been proposed to allow SMTP servers to *authenticate* the messages sent by their users **RFC 4870** **RFC 4871** .

The Post Office Protocol

When the first versions of SMTP were designed, the Internet was composed of minicomputers that were used by an entire university department or research lab. These minicomputers were used by many users at the same time. Email was mainly used to send messages from a user on a given host to another user on a remote host. At that time, SMTP was the only protocol involved in the delivery of the emails as all hosts attached to the network were running an SMTP server. On such hosts, an email destined to local users was delivered by placing the email in a special directory or file owned by the user. However, the introduction of personal computers in the 1980s, changed this environment. Initially, users of these personal computers used applications such as *telnet* to open a remote session on the local *minicomputer* to read their email. This was not user-friendly. A better solution appeared with the development of user friendly email client applications on personal computers. Several protocols were designed to allow these client applications to retrieve the email messages destined to a user from his/her server. Two of these protocols became popular and are still used today. The Post Office Protocol (POP), defined in **RFC 1939**, is the simplest one. It allows a client to download all the messages destined to a given user from his/her email server. We describe POP briefly in this section. The second protocol is the Internet Message Access Protocol (IMAP), defined in **RFC 3501**. IMAP is more powerful, but also more complex than POP. IMAP was designed to allow client applications to efficiently access in real-time to messages stored in various folders on servers. IMAP

[15] This implies that a valid email message cannot contain a line with one dot followed by *CR* and *LF*. If a user types such a line in an email, his email client will automatically add a space character before or after the dot when sending the message over SMTP.

assumes that all the messages of a given user are stored on a server and provides the functions that are necessary to search, download, delete or filter messages.

POP is another example of a simple line-based protocol. POP runs above the bytestream service. A POP server usually listens to port 110. A POP session is composed of three parts : an *authorisation* phase during which the server verifies the client's credential, a *transaction* phase during which the client downloads messages and an *update* phase that concludes the session. The client sends commands and the server replies are prefixed by *+OK* to indicate a successful command or by *-ERR* to indicate errors.

When a client opens a transport connection with the POP server, the latter sends as banner an ASCII-line starting with *+OK*. The POP session is at that time in the *authorisation* phase. In this phase, the client can send its username (resp. password) with the *USER* (resp. *PASS*) command. The server replies with *+OK* if the username (resp. password) is valid and *-ERR* otherwise.

Once the username and password have been validated, the POP session enters in the *transaction* phase. In this phase, the client can issue several commands. The *STAT* command is used to retrieve the status of the server. Upon reception of this command, the server replies with a line that contains *+OK* followed by the number of messages in the mailbox and the total size of the mailbox in bytes. The *RETR* command, followed by a space and an integer, is used to retrieve the nth message of the mailbox. The *DELE* command is used to mark for deletion the nth message of the mailbox.

Once the client has retrieved and possibly deleted the emails contained in the mailbox, it must issue the *QUIT* command. This command terminates the POP session and allows the server to delete all the messages that have been marked for deletion by using the *DELE* command.

The figure below provides a simple POP session. All lines prefixed with *C:* (resp. *S:*) are sent by the client (resp. server).

```
S:    +OK POP3 server ready
C:    USER alice
S:    +OK
C     PASS 12345pass
S:    +OK alice's maildrop has 2 messages (620 octets)
C:    STAT
S:    +OK 2 620
C:    LIST
S:    +OK 2 messages (620 octets)
S:    1 120
S:    2 500
S:    .
C:    RETR 1
S:    +OK 120 octets
S:    <the POP3 server sends message 1>
S:    .
C:    DELE 1
S:    +OK message 1 deleted
C:    QUIT
S:    +OK POP3 server signing off (1 message left)
```

In this example, a POP client contacts a POP server on behalf of the user named *alice*. Note that in this example, Alice's password is sent in clear by the client. This implies that if someone is able to capture the packets sent by Alice, he will know Alice's password [16]. Then Alice's client issues the *STAT* command to know the number of messages that are stored in her mailbox. It then retrieves and deletes the first message of the mailbox.

3.2.3 The HyperText Transfer Protocol

In the early days of the Internet was mainly used for remote terminal access with telnet, email and file transfer. The default file transfer protocol, *FTP*, defined in **RFC 959** was widely used and *FTP* clients and servers are still included in most operating systems.

16 **RFC 1939** defines the APOP authentication scheme that is not vulnerable to such attacks.

Many *FTP* clients offer a user interface similar to a Unix shell and allow the client to browse the file system on the server and to send and retrieve files. *FTP* servers can be configured in two modes :

- authenticated : in this mode, the ftp server only accepts users with a valid user name and password. Once authenticated, they can access the files and directories according to their permissions
- anonymous : in this mode, clients supply the *anonymous* userid and their email address as password. These clients are granted access to a special zone of the file system that only contains public files.

ftp was very popular in the 1990s and early 2000s, but today it has mostly been superseded by more recent protocols. Authenticated access to files is mainly done by using the Secure Shell (ssh) protocol defined in **RFC 4251** and supported by clients such as scp or sftp. Nowadays, anonymous access is mainly provided by web protocols.

In the late 1980s, high energy physicists working at CERN had to efficiently exchange documents about their ongoing and planned experiments. Tim Berners-Lee evaluated several of the documents sharing techniques that were available at that time [B1989]. As none of the existing solutions met CERN's requirements, they choose to develop a completely new document sharing system. This system was initially called the *mesh*, but was quickly renamed the *world wide web*. The starting point for the *world wide web* are hypertext documents. An hypertext document is a document that contains references (hyperlinks) to other documents that the reader can immediately access. Hypertext was not invented for the world wide web. The idea of hypertext documents was proposed in 1945 [Bush1945] and the first experiments were done during the 1960s [Nelson1965] [Myers1998] . Compared to the hypertext documents that were used in the late 1980s, the main innovation introduced by the *world wide web* was to allow hyperlinks to reference documents stored on remote machines.

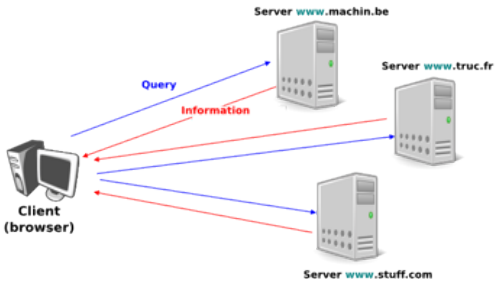

Figure 3.17: World-wide web clients and servers

A document sharing system such as the *world wide web* is composed of three important parts.

1. A standardised addressing scheme that allows unambiguous identification of documents
2. A standard document format : the HyperText Markup Language
3. A standardised protocol that facilitates efficient retrieval of documents stored on a server

---

Note: Open standards and open implementations

Open standards have, and are still playing a key role in the success of the *world wide web* as we know it today. Without open standards, the world wide web would never have reached its current size. In addition to open standards, another important factor for the success of the web was the availability of open and efficient implementations of these standards. When CERN started to work on the *web*, their objective was to build a running system that could be used by physicists. They developed open-source implementations of the first web servers and web clients. These open-source implementations were powerful and could be used as is, by institutions willing to

share information on the web. They were also extended by other developers who contributed to new features. For example, NCSA added support for images in their Mosaic browser that was eventually used to create Netscape Communications.

The first components of the *world wide web* are the Uniform Resource Identifiers (URI), defined in **RFC 3986**. A URI is a character string that unambiguously identifies a resource on the world wide web. Here is a subset of the BNF for URIs

```
URI          = scheme ":" "//" authority path [ "?" query ] [ "#" fragment ]
scheme       = ALPHA *( ALPHA / DIGIT / "+" / "-" / "." )
authority    = [ userinfo "@" ] host [ ":" port ]
query        = *( pchar / "/" / "?" )
fragment     = *( pchar / "/" / "?" )
pchar          = unreserved / pct-encoded / sub-delims / ":" / "@"
query          = *( pchar / "/" / "?" )
fragment       = *( pchar / "/" / "?" )
pct-encoded    = "%" HEXDIG HEXDIG
unreserved     = ALPHA / DIGIT / "-" / "." / "_" / "~"
reserved       = gen-delims / sub-delims
gen-delims     = ":" / "/" / "?" / "#" / "[" / "]" / "@"
sub-delims     = "!" / "$" / "&" / "'" / "(" / ")" / "*" / "+" / "," / ";" / "="
```

The first component of a URI is its *scheme*. A *scheme* can be seen as a selector, indicating the meaning of the fields after it. In practice, the scheme often identifies the application-layer protocol that must be used by the client to retrieve the document, but it is not always the case. Some schemes do not imply a protocol at all and some do not indicate a retrievable document [17]. The most frequent scheme is *http that will be described later. A URI scheme can be defined for almost any application layer protocol [#furilist]_. The characters ':* and *//* follow the *scheme* of any URI.

The second part of the URI is the *authority*. With retrievable URI, this includes the DNS name or the IP address of the server where the document can be retrieved using the protocol specified via the *scheme*. This name can be preceded by some information about the user (e.g. a user name) who is requesting the information. Earlier definitions of the URI allowed the specification of a user name and a password before the @ character (**RFC 1738**), but this is now deprecated as placing a password inside a URI is insecure. The host name can be followed by the semicolon character and a port number. A default port number is defined for some protocols and the port number should only be included in the URI if a non-default port number is used (for other protocols, techniques like service DNS records are used).

The third part of the URI is the path to the document. This path is structured as filenames on a Unix host (but it does not imply that the files are indeed stored this way on the server). If the path is not specified, the server will return a default document. The last two optional parts of the URI are used to provide a query and indicate a specific part (e.g. a section in an article) of the requested document. Sample URIs are shown below.

```
http://tools.ietf.org/html/rfc3986.html
mailto:infobot@example.com?subject=current-issue
http://docs.python.org/library/basehttpserver.html?highlight=http#BaseHTTPServer.BaseHTTPRequestHa
telnet://[2001:6a8:3080:3::2]:80/
ftp://cnn.example.com&story=breaking_news@10.0.0.1/top_story.htm
```

The first URI corresponds to a document named *rfc3986.html* that is stored on the server named *tools.ietf.org* and can be accessed by using the *http* protocol on its default port. The second URI corresponds to an email message, with subject *current-issue*, that will be sent to user *infobot* in domain *example.com*. The *mailto:* URI scheme is defined in **RFC 6068**. The third URI references the portion *BaseHTTPServer.BaseHTTPRequestHandler* of the document *basehttpserver.html* that is stored in the *library* directory on server *docs.python.org*. This document can be retrieved by using the *http* protocol. The query *highlight=http* is associated to this URI. The fourth example is a server that operates the telnet protocol, uses IPv6 address *2001:6a8:3080:3::2* and is reachable on port 80. The last URI is somewhat special. Most users will assume that it corresponds to a document stored on the *cnn.example.com*

[17] An example of a non-retrievable URI is *urn:isbn:0-380-81593-1* which is an unique identifier for a book, through the urn scheme (see **RFC 3187**). Of course, any URI can be make retrievable via a dedicated server or a new protocol but this one has no explicit protocol. Same thing for the scheme tag (see **RFC 4151**), often used in Web syndication (see **RFC 4287** about the Atom syndication format). Even when the scheme is retrievable (for instance with http'), it is often used only as an identifier, not as a way to get a resource. See http://norman.walsh.name/2006/07/25/namesAndAddresses for a good explanation.

server. However, to parse this URI, it is important to remember that the @ character is used to separate the user name from the host name in the authorisation part of a URI. This implies that the URI points to a document named *top_story.htm* on host having IPv4 address *10.0.0.1*. The document will be retrieved by using the *ftp* protocol with the user name set to *cnn.example.com&story=breaking_news*.

The second component of the *word wide web* is the HyperText Markup Language (HTML). HTML defines the format of the documents that are exchanged on the *web*. The first version of HTML was derived from the Standard Generalized Markup Language (SGML) that was standardised in 1986 by *ISO*. SGML was designed to allow large project documents in industries such as government, law or aerospace to be shared efficiently in a machine-readable manner. These industries require documents to remain readable and editable for tens of years and insisted on a standardised format supported by multiple vendors. Today, SGML is no longer widely used beyond specific applications, but its descendants including *HTML* and *XML* are now widespread.

A markup language is a structured way of adding annotations about the formatting of the document within the document itself. Example markup languages include troff, which is used to write the Unix man pages or Latex. HTML uses markers to annotate text and a document is composed of *HTML elements*. Each element is usually composed of three items: a start tag that potentially includes some specific attributes, some text (often including other elements), and an end tag. A HTML tag is a keyword enclosed in angle brackets. The generic form of a HTML element is

```
<tag>Some text to be displayed</tag>
```

More complex HTML elements can also include optional attributes in the start tag

```
<tag attribute1="value1" attribute2="value2">some text to be displayed</tag>
```

The HTML document shown below is composed of two parts : a header, delineated by the *<head>* and *</head>* markers, and a body (between the *<body>* and *</body>* markers). In the example below, the header only contains a title, but other types of information can be included in the header. The body contains an image, some text and a list with three hyperlinks. The image is included in the web page by indicating its URI between brackets inside the *<img src="...">* marker. The image can, of course, reside on any server and the client will automatically download it when rendering the web page. The *<h1>...</h1>* marker is used to specify the first level of headings. The *<ul>* marker indicates an unnumbered list while the *<li>* marker indicates a list item. The *text* indicates a hyperlink. The *text* will be underlined in the rendered web page and the client will fetch the specified URI when the user clicks on the link.

Figure 3.18: A simple HTML page

Additional details about the various extensions to HTML may be found in the official specifications maintained by W3C.

The third component of the *world wide web* is the HyperText Transport Protocol (HTTP). HTTP is a text-based protocol, in which the client sends a request and the server returns a response. HTTP runs above the bytestream service and HTTP servers listen by default on port *80*. The design of HTTP has largely been inspired by the Internet email protocols. Each HTTP request contains three parts :

- a *method* , that indicates the type of request, a URI, and the version of the HTTP protocol used by the client

- a *header* , that is used by the client to specify optional parameters for the request. An empty line is used to mark the end of the header
- an optional MIME document attached to the request

The response sent by the server also contains three parts :

- a *status line* , that indicates whether the request was successful or not
- a *header* , that contains additional information about the response. The response header ends with an empty line.
- a MIME document

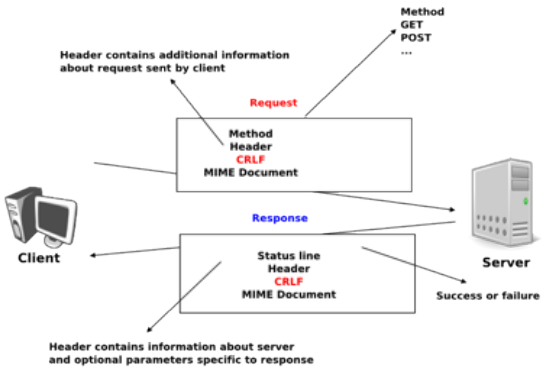

Figure 3.19: HTTP requests and responses

Several types of method can be used in HTTP requests. The three most important ones are :

- the *GET* method is the most popular one. It is used to retrieve a document from a server. The *GET* method is encoded as *GET* followed by the path of the URI of the requested document and the version of HTTP used by the client. For example, to retrieve the http://www.w3.org/MarkUp/ URI, a client must open a TCP on port *80* with host *www.w3.org* and send a HTTP request containing the following line

  GET /MarkUp/ HTTP/1.0

- – the *HEAD* method is a variant of the *GET* method that allows the retrieval of the header lines for a given URI without retrieving the entire document. It can be used by a client to verify if a document exists, for instance.
- the *POST* method can be used by a client to send a document to a server. The sent document is attached to the HTTP request as a MIME document.

HTTP clients and servers can include many different HTTP headers in HTTP requests and responses. Each HTTP header is encoded as a single ASCII-line terminated by *CR* and *LF*. Several of these headers are briefly described below. A detailed discussion of all standard headers may be found in **RFC 1945**. The MIME headers can appear in both HTTP requests and HTTP responses.

- the *Content-Length:* header is the *MIME* header that indicates the length of the MIME document in bytes.
- the *Content-Type:* header is the *MIME* header that indicates the type of the attached MIME document. HTML pages use the *text/html* type.
- the *Content-Encoding:* header indicates how the *MIME document* has been encoded. For example, this header would be set to *x-gzip* for a document compressed using the gzip software.

RFC 1945 and **RFC 2616** define headers that are specific to HTTP responses. These server headers include :

- the *Server:* header indicates the version of the web server that has generated the HTTP response. Some servers provide information about their software release and optional modules that they use. For security reasons, some system administrators disable these headers to avoid revealing too much information about their server to potential attackers.
- the *Date:* header indicates when the HTTP response has been produced by the server.
- the *Last-Modified:* header indicates the date and time of the last modification of the document attached to the HTTP response.

Similarly, the following header lines can only appear inside HTTP requests sent by a client :

- the *User-Agent:* header provides information about the client that has generated the HTTP request. Some servers analyse this header line and return different headers and sometimes different documents for different user agents.
- the *If-Modified-Since:* header is followed by a date. It enables clients to cache in memory or on disk the recent or most frequently used documents. When a client needs to request a URI from a server, it first checks whether the document is already in its cache. If it is, the client sends a HTTP request with the *If-Modified-Since:* header indicating the date of the cached document. The server will only return the document attached to the HTTP response if it is newer than the version stored in the client's cache.
- the *Referrer:* header is followed by a URI. It indicates the URI of the document that the client visited before sending this HTTP request. Thanks to this header, the server can know the URI of the document containing the hyperlink followed by the client, if any. This information is very useful to measure the impact of advertisements containing hyperlinks placed on websites.
- the *Host:* header contains the fully qualified domain name of the URI being requested.

Note: The importance of the *Host:* header line

The first version of HTTP did not include the *Host:* header line. This was a severe limitation for web hosting companies. For example consider a web hosting company that wants to serve both *web.example.com* and *www.example.net* on the same physical server. Both web sites contain a */index.html* document. When a client sends a request for either *http://web.example.com/index.html* or *http://www.example.net/index.html*, the HTTP 1.0 request contains the following line :

```
GET /index.html HTTP/1.0
```

By parsing this line, a server cannot determine which *index.html* file is requested. Thanks to the *Host:* header line, the server knows whether the request is for *http://web.example.com/index.html* or *http://www.dummy.net/index.html*. Without the *Host:* header, this is impossible. The *Host:* header line allowed web hosting companies to develop their business by supporting a large number of independent web servers on the same physical server.

The status line of the HTTP response begins with the version of HTTP used by the server (usually *HTTP/1.0* defined in **RFC 1945** or *HTTP/1.1* defined in **RFC 2616**) followed by a three digit status code and additional information in English. HTTP status codes have a similar structure as the reply codes used by SMTP.

- All status codes starting with digit *2* indicate a valid response. *200 Ok* indicates that the HTTP request was successfully processed by the server and that the response is valid.
- All status codes starting with digit *3* indicate that the requested document is no longer available on the server. *301 Moved Permanently* indicates that the requested document is no longer available on this server. A *Location:* header containing the new URI of the requested document is inserted in the HTTP response. *304 Not Modified* is used in response to an HTTP request containing the *If-Modified-Since:* header. This status line is used by the server if the document stored on the server is not more recent than the date indicated in the *If-Modified-Since:* header.
- All status codes starting with digit *4* indicate that the server has detected an error in the HTTP request sent by the client. *400 Bad Request* indicates a syntax error in the HTTP request. *404 Not Found* indicates that the requested document does not exist on the server.

- All status codes starting with digit *5* indicate an error on the server. *500 Internal Server Error* indicates that the server could not process the request due to an error on the server itself.

In both the HTTP request and the HTTP response, the MIME document refers to a representation of the document with the MIME headers indicating the type of document and its size.

As an illustration of HTTP/1.0, the transcript below shows a HTTP request for http://www.ietf.org and the corresponding HTTP response. The HTTP request was sent using the curl command line tool. The *User-Agent:* header line contains more information about this client software. There is no MIME document attached to this HTTP request, and it ends with a blank line.

```
GET / HTTP/1.0
User-Agent: curl/7.19.4 (universal-apple-darwin10.0) libcurl/7.19.4 OpenSSL/0.9.8l zlib/1.2.3
Host: www.ietf.org
```

The HTTP response indicates the version of the server software used with the modules included. The *Last-Modified:* header indicates that the requested document was modified about one week before the request. A HTML document (not shown) is attached to the response. Note the blank line between the header of the HTTP response and the attached MIME document. The *Server:* header line has been truncated in this output.

```
HTTP/1.1 200 OK
Date: Mon, 15 Mar 2010 13:40:38 GMT
Server: Apache/2.2.4 (Linux/SUSE) mod_ssl/2.2.4 OpenSSL/0.9.8e (truncated)
Last-Modified: Tue, 09 Mar 2010 21:26:53 GMT
Content-Length: 17019
Content-Type: text/html

<!DOCTYPE HTML PUBLIC .../HTML>
```

HTTP was initially designed to share self-contained text documents. For this reason, and to ease the implementation of clients and servers, the designers of HTTP chose to open a TCP connection for each HTTP request. This implies that a client must open one TCP connection for each URI that it wants to retrieve from a server as illustrated on the figure below. For a web page containing only text documents this was a reasonable design choice as the client usually remains idle while the (human) user is reading the retrieved document.

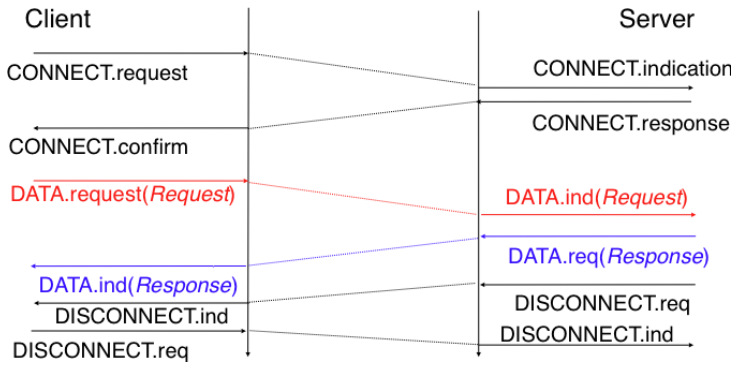

Figure 3.20: HTTP 1.0 and the underlying TCP connection

However, as the web evolved to support richer documents containing images, opening a TCP connection for each URI became a performance problem [Mogul1995]. Indeed, besides its HTML part, a web page may include dozens of images or more. Forcing the client to open a TCP connection for each component of a web page has two important drawbacks. First, the client and the server must exchange packets to open and close a TCP connection as we will see later. This increases the network overhead and the total delay of completely retrieving all the components of a web page. Second, a large number of established TCP connections may be a performance bottleneck on servers.

This problem was solved by extending HTTP to support persistent TCP connections **RFC 2616**. A persistent connection is a TCP connection over which a client may send several HTTP requests. This is illustrated in the figure below.

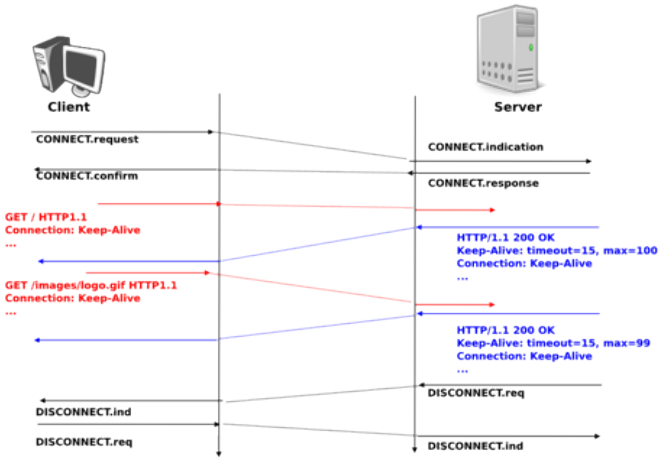

Figure 3.21: HTTP 1.1 persistent connections

To allow the clients and servers to control the utilisation of these persistent TCP connections, HTTP 1.1 **RFC 2616** defines several new HTTP headers :

- The *Connection:* header is used with the *Keep-Alive* argument by the client to indicate that it expects the underlying TCP connection to be persistent. When this header is used with the *Close* argument, it indicates that the entity that sent it will close the underlying TCP connection at the end of the HTTP response.
- The *Keep-Alive:* header is used by the server to inform the client about how it agrees to use the persistent connection. A typical *Keep-Alive:* contains two parameters : the maximum number of requests that the server agrees to serve on the underlying TCP connection and the timeout (in seconds) after which the server will close an idle connection

The example below shows the operation of HTTP/1.1 over a persistent TCP connection to retrieve three URIs stored on the same server. Once the connection has been established, the client sends its first request with the *Connection: keep-alive* header to request a persistent connection.

```
GET / HTTP/1.1
Host: www.kame.net
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_2; en-us)
Connection: keep-alive
```

The server replies with the *Connection: Keep-Alive* header and indicates that it accepts a maximum of 100 HTTP requests over this connection and that it will close the connection if it remains idle for 15 seconds.

```
HTTP/1.1 200 OK
Date: Fri, 19 Mar 2010 09:23:37 GMT
Server: Apache/2.0.63 (FreeBSD) PHP/5.2.12 with Suhosin-Patch
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Length: 3462
Content-Type: text/html

<html>...   </html>
```

The client sends a second request for the style sheet of the retrieved web page.

```
GET /style.css HTTP/1.1
Host: www.kame.net
Referer: http://www.kame.net/
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_2; en-us)
Connection: keep-alive
```

The server replies with the requested style sheet and maintains the persistent connection. Note that the server only accepts 99 remaining HTTP requests over this persistent connection.

```
HTTP/1.1 200 OK
Date: Fri, 19 Mar 2010 09:23:37 GMT
Server: Apache/2.0.63 (FreeBSD) PHP/5.2.12 with Suhosin-Patch
Last-Modified: Mon, 10 Apr 2006 05:06:39 GMT
Content-Length: 2235
Keep-Alive: timeout=15, max=99
Connection: Keep-Alive
Content-Type: text/css

...
```

Then the client automatically requests the web server's icon [18] , that could be displayed by the browser. This server does not contain such URI and thus replies with a *404* HTTP status. However, the underlying TCP connection is not closed immediately.

```
GET /favicon.ico HTTP/1.1
Host: www.kame.net
Referer: http://www.kame.net/
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_2; en-us)
Connection: keep-alive

HTTP/1.1 404 Not Found
Date: Fri, 19 Mar 2010 09:23:40 GMT
Server: Apache/2.0.63 (FreeBSD) PHP/5.2.12 with Suhosin-Patch
Content-Length: 318
Keep-Alive: timeout=15, max=98
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN"> ...
```

As illustrated above, a client can send several HTTP requests over the same persistent TCP connection. However, it is important to note that all of these HTTP requests are considered to be independent by the server. Each HTTP request must be self-contained. This implies that each request must include all the header lines that are required by the server to understand the request. The independence of these requests is one of the important design choices of HTTP. As a consequence of this design choice, when a server processes a HTTP request, it doesn't' use any other information than what is contained in the request itself. This explains why the client adds its *User-Agent:* header in all of the HTTP requests it sends over the persistent TCP connection.

However, in practice, some servers want to provide content tuned for each user. For example, some servers can provide information in several languages or other servers want to provide advertisements that are targeted to different types of users. To do this, servers need to maintain some information about the preferences of each user and use this information to produce content matching the user's preferences. HTTP contains several mechanisms that enable to solve this problem. We discuss three of them below.

A first solution is to force the users to be authenticated. This was the solution used by *FTP* to control the files that each user could access. Initially, user names and passwords could be included inside URIs **RFC 1738**. However, placing passwords in the clear in a potentially publicly visible URI is completely insecure and this usage has now been deprecated **RFC 3986**. HTTP supports several extension headers **RFC 2617** that can be used by a server to request the authentication of the client by providing his/her credentials. However, user names and passwords have not been popular on web servers as they force human users to remember one user name and one password per server. Remembering a password is acceptable when a user needs to access protected content, but users will not accept the need for a user name and password only to receive targeted advertisements from the web sites that they visit.

A second solution to allow servers to tune that content to the needs and capabilities of the user is to rely on the different types of *Accept-** HTTP headers. For example, the *Accept-Language:* can be used by the client to

[18] Favorite icons are small icons that are used to represent web servers in the toolbar of Internet browsers. Microsoft added this feature in their browsers without taking into account the W3C standards. See http://www.w3.org/2005/10/howto-favicon for a discussion on how to cleanly support such favorite icons.

indicate its preferred languages. Unfortunately, in practice this header is usually set based on the default language of the browser and it is not possible for a user to indicate the language it prefers to use by selecting options on each visited web server.

The third, and widely adopted, solution are HTTP cookies. HTTP cookies were initially developed as a private extension by Netscape. They are now part of the standard **RFC 6265**. In a nutshell, a cookie is a short string that is chosen by a server to represent a given client. Two HTTP headers are used : *Cookie:* and *Set-Cookie:*. When a server receives an HTTP request from a new client (i.e. an HTTP request that does not contain the *Cookie:* header), it generates a cookie for the client and includes it in the *Set-Cookie:* header of the returned HTTP response. The *Set-Cookie:* header contains several additional parameters including the domain names for which the cookie is valid. The client stores all received cookies on disk and every time it sends a HTTP request, it verifies whether it already knows a cookie for this domain. If so, it attaches the *Cookie:* header to the HTTP request. This is illustrated in the figure below with HTTP 1.1, but cookies also work with HTTP 1.0.

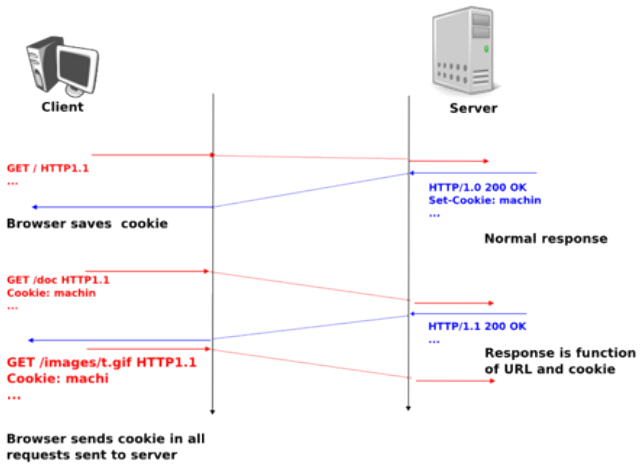

Figure 3.22: HTTP cookies

Note: Privacy issues with HTTP cookies

The HTTP cookies introduced by Netscape are key for large e-commerce websites. However, they have also raised many discussions concerning their potential misuses. Consider *ad.com*, a company that delivers lots of advertisements on web sites. A web site that wishes to include *ad.com*'s advertisements next to its content will add links to *ad.com* inside its HTML pages. If *ad.com* is used by many web sites, *ad.com* could be able to track the interests of all the users that visit its client websites and use this information to provide targeted advertisements. Privacy advocates have even sued online advertisement companies to force them to comply with the privacy regulations. More recent related technologies also raise privacy concerns

---

3.3 Writing simple networked applications

Networked applications were usually implemented by using the *socket API*. This API was designed when TCP/IP was first implemented in the Unix BSD operating system [Sechrest] [LFJLMT], and has served as the model for many APIs between applications and the networking stack in an operating system. Although the socket API is very popular, other APIs have also been developed. For example, the STREAMS API has been added to several Unix System V variants [Rago1993]. The socket API is supported by most programming languages and several textbooks have been devoted to it. Users of the C language can consult [DC2009], [Stevens1998], [SFR2004] or [Kerrisk2010]. The Java implementation of the socket API is described in [CD2008] and in the Java tutorial. In this section, we will use the python implementation of the socket API to illustrate the key concepts. Additional information about this API may be found in the socket section of the python documentation .

The socket API is quite low-level and should be used only when you need a complete control of the network access. If your application simply needs, for instance, to retrieve data with HTTP, there are much simpler and higher-level APIs.

A detailed discussion of the socket API is outside the scope of this section and the references cited above provide a detailed discussion of all the details of the socket API. As a starting point, it is interesting to compare the socket API with the service primitives that we have discussed in the previous chapter. Let us first consider the connectionless service that consists of the following two primitives :

- *DATA.request(destination,message)* is used to send a message to a specified destination. In this socket API, this corresponds to the `send` method.
- *DATA.indication(message)* is issued by the transport service to deliver a message to the application. In the socket API, this corresponds to the return of the `recv` method that is called by the application.

The *DATA* primitives are exchanged through a service access point. In the socket API, the equivalent to the service access point is the *socket*. A *socket* is a data structure which is maintained by the networking stack and is used by the application every time it needs to send or receive data through the networking stack. The *socket* method in the python API takes two main arguments :

- an *address family* that specifies the type of address family and thus the underlying networking stack that will be used with the socket. This parameter can be either `socket.AF_INET` or `socket.AF_INET6`. `socket.AF_INET`, which corresponds to the TCP/IPv4 protocol stack is the default. `socket.AF_INET6` corresponds to the TCP/IPv6 protocol stack.
- a *type* indicates the type of service which is expected from the networking stack. `socket.STREAM` (the default) corresponds to the reliable bytestream connection-oriented service. `socket.DGRAM` corresponds to the connectionless service.

A simple client that sends a request to a server is often written as follows in descriptions of the socket API.

```
# A simple client of the connectionless service
import socket
import sys
HOSTIP=sys.argv[1]
PORT=int(sys.argv[2])
MSG="Hello, World!"
s = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )
s.sendto( MSG, (HOSTIP, PORT) )
```

A typical usage of this application would be

```
python client.py 127.0.0.1 12345
```

where `127.0.0.1` is the IPv4 address of the host (in this case the localhost) where the server is running and `12345` the port of the server.

The first operation is the creation of the `socket`. Two parameters must be specified while creating a `socket`. The first parameter indicates the address family and the second the socket type. The second operation is the transmission of the message by using `sendto` to the server. It should be noted that `sendto` takes as arguments the message to be transmitted and a tuple that contains the IPv4 address of the server and its port number.

The code shown above supports only the TCP/IPv4 protocol stack. To use the TCP/IPv6 protocol stack the `socket` must be created by using the `socket.AF_INET6` address family. Forcing the application developer to select TCP/IPv4 or TCP/IPv6 when creating a `socket` is a major hurdle for the deployment and usage of TCP/IPv6 in the global Internet [Cheshire2010]. While most operating systems support both TCP/IPv4 and TCP/IPv6, many applications still only use TCP/IPv4 by default. In the long term, the `socket` API should be able to handle TCP/IPv4 and TCP/IPv6 transparently and should not force the application developer to always specify whether it uses TCP/IPv4 or TCP/IPv6.

Another important issue with the socket API as supported by python is that it forces the application to deal with IP addresses instead of dealing directly with domain names. This limitation dates from the early days of the `socket` API in Unix 4.2BSD. At that time, the DNS was not widely available and only IP addresses could be used. Most applications rely on DNS names to interact with servers and this utilisation of the DNS plays a very important role to scale web servers and content distribution networks. To use domain names, the application needs

to perform the DNS resolution by using the `getaddrinfo` method. This method queries the DNS and builds the `sockaddr` data structure which is used by other methods of the socket API. In python, `getaddrinfo` takes several arguments :

- a *name* that is the domain name for which the DNS will be queried
- an optional *port number* which is the port number of the remote server
- an optional *address family* which indicates the address family used for the DNS request. `socket.AF_INET` (resp. `socket.AF_INET6`) indicates that an IPv4 (IPv6) address is expected. Furthermore, the python socket API allows an application to use `socket.AF_UNSPEC` to indicate that it is able to use either IPv4 or IPv6 addresses.
- an optional *socket type* which can be either `socket.SOCK_DGRAM` or `socket.SOCK_STREAM`

In today's Internet hosts that are capable of supporting both IPv4 and IPv6, all applications should be able to handle both IPv4 and IPv6 addresses. When used with the `socket.AF_UNSPEC` parameter, the `socket.getaddrinfo` method returns a list of tuples containing all the information to create a `socket`.

```
import socket
socket.getaddrinfo('www.example.net',80,socket.AF_UNSPEC,socket.SOCK_STREAM)
[ (30, 1, 6, '', ('2001:db8:3080:3::2', 80, 0, 0)),
  (2, 1, 6, '', ('203.0.113.225', 80))]
```

In the example above, `socket.getaddrinfo` returns two tuples. The first one corresponds to the `sockaddr` containing the IPv6 address of the remote server and the second corresponds to the IPv4 information. Due to some peculiarities of IPv6 and IPv4, the format of the two tuples is not exactly the same, but the key information in both cases are the network layer address (`2001:db8:3080:3::2` and `203.0.113.225`) and the port number (`80`). The other parameters are seldom used.

`socket.getaddrinfo` can be used to build a simple client that queries the DNS and contact the server by using either IPv4 or IPv6 depending on the addresses returned by the `socket.getaddrinfo` method. The client below iterates over the list of addresses returned by the DNS and sends its request to the first destination address for which it can create a `socket`. Other strategies are of course possible. For example, a host running in an IPv6 network might prefer to always use IPv6 when IPv6 is available [19]. Another example is the happy eyeballs approach which is being discussed within the IETF [WY2011]. For example, [WY2011] mentions that some web browsers try to use the first address returned by `socket.getaddrinfo`. If there is no answer within some small delay (e.g. 300 milliseconds), the second address is tried.

```
import socket
import sys
HOSTNAME=sys.argv[1]
PORT=int(sys.argv[2])
MSG="Hello, World!"
for a in socket.getaddrinfo(HOSTNAME, PORT, socket.AF_UNSPEC,socket.SOCK_DGRAM,0, socket.AI_PASSIV
    address_family,sock_type,protocol,canonicalname, sockaddr=a
    try:
        s = socket.socket(address_family, sock_type)
    except socket.error:
        s = None
        print "Could not create socket"
        continue
    if s is not None:
        s.sendto(MSG, sockaddr)
        break
```

Now that we have described the utilisation of the socket API to write a simple client using the connectionless transport service, let us have a closer look at the reliable byte stream transport service. As explained above, this service is invoked by creating a `socket` of type `socket.SOCK_STREAM`. Once a socket has been created, a client will typically connect to the remote server, send some data, wait for an answer and eventually close the connection. These operations are performed by calling the following methods :

[19] Most operating systems today by default prefer to use IPv6 when the DNS returns both an IPv4 and an IPv6 address for a name. See http://ipv6int.net/systems/ for more detailed information.

- `socket.connect` : this method takes a `sockaddr` data structure, typically returned by `socket.getaddrinfo`, as argument. It may fail and raise an exception if the remote server cannot be reached.
- `socket.send` : this method takes a string as argument and returns the number of bytes that were actually sent. The string will be transmitted as a sequence of consecutive bytes to the remote server. Applications are expected to check the value returned by this method and should resend the bytes that were not send.
- `socket.recv` : this method takes an integer as argument that indicates the size of the buffer that has been allocated to receive the data. An important point to note about the utilisation of the `socket.recv` method is that as it runs above a bytestream service, it may return any amount of bytes (up to the size of the buffer provided by the application). The application needs to collect all the received data and there is no guarantee that some data sent by the remote host by using a single call to the `socket.send` method will be received by the destination with a single call to the `socket.recv` method.
- `socket.shutdown` : this method is used to release the underlying connection. On some platforms, it is possible to specify the direction of transfer to be released (e.g. `socket.SHUT_WR` to release the outgoing direction or `socket.SHUT_RDWR` to release both directions).
- `socket.close`: this method is used to close the socket. It calls `socket.shutdown` if the underlying connection is still open.

With these methods, it is now possible to write a simple HTTP client. This client operates over both IPv6 and IPv4 and writes the homepage of the remote server on the standard output. It also reports the number of `socket.recv` calls that were used to retrieve the homepage [20] .

```
#!/usr/bin/python
# A simple http client that retrieves the first page of a web site

import socket, sys

if len(sys.argv)!=3 and len(sys.argv)!=2:
    print "Usage : ",sys.argv[0]," hostname [port]"

hostname = sys.argv[1]
if len(sys.argv)==3 :
    port=int(sys.argv[2])
else:
    port = 80

READBUF=16384   # size of data read from web server
s=None

for res in socket.getaddrinfo(hostname, port, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    # create socket
    try:
        s = socket.socket(af, socktype, proto)
    except socket.error:
        s = None
        continue
    # connect to remote host
    try:
        print "Trying "+sa[0]
        s.connect(sa)
    except socket.error, msg:
        # socket failed
        s.close()
        s = None
        continue
    if s :
```

[20] Experiments with the client indicate that the number of *socket.recv* calls can vary at each run. There are various factors that influence the number of such calls that are required to retrieve some information from a server. We'll discuss some of them after having explained the operation of the underlying transport protocol.

```
        print "Connected to "+sa[0]
        s.send('GET / HTTP/1.1\r\nHost:'+hostname+'\r\n\r\n')
        finished=False
        count=0
        while not finished:
            data=s.recv(READBUF)
            count=count+1
            if len(data)!=0:
                print repr(data)
            else:
                finished=True
        s.shutdown(socket.SHUT_WR)
        s.close()
        print "Data was received in ",count," recv calls"
        break
```

As mentioned above, the socket API is very low-level. This is the interface to the transport service. For a common and simple task, like retrieving a document from the Web, there are much simpler solutions. For example, the python standard library includes several high-level APIs to implementations of various application layer protocols including HTTP. For example, the httplib module can be used to easily access documents via HTTP.

```
#!/usr/bin/python
# A simple http client that retrieves the first page of a web site, using
# the standard httplib library

import httplib, sys

if len(sys.argv)!=3 and len(sys.argv)!=2:
 print "Usage : ",sys.argv[0]," hostname [port]"
 sys.exit(1)

path = '/'
hostname = sys.argv[1]
if len(sys.argv)==3 :
 port = int(sys.argv[2])
else:
 port = 80

conn = httplib.HTTPConnection(hostname, port)
conn.request("GET", path)
r = conn.getresponse()
print "Response is %i (%s)" % (r.status, r.reason)
print r.read()
```

Another module, urllib2 allows the programmer to directly use URLs. This is much more simpler than directly using sockets.

But simplicity is not the only advantage of using high-level libraries. They allow the programmer to manipulate higher-level concepts (e.g. *I want the content pointed by this URL*) but also include many features such as transparent support for the utilisation of *TLS* or IPv6.

The second type of applications that can be written by using the socket API are the servers. A server is typically runs forever waiting to process requests coming from remote clients. A server using the connectionless will typically start with the creation of a *socket* with the `socket.socket`. This socket can be created above the TCP/IPv4 networking stack (`socket.AF_INET`) or the TCP/IPv6 networking stack (`socket.AF_INET6`), but not both by default. If a server is willing to use the two networking stacks, it must create two threads, one to handle the TCP/IPv4 socket and the other to handle the TCP/IPv6 socket. It is unfortunately impossible to define a socket that can receive data from both networking stacks at the same time with the python socket API.

A server using the connectionless service will typically use two methods from the socket API in addition to those that we have already discussed.

- `socket.bind` is used to bind a socket to a port number and optionally an IP address. Most servers will bind their socket to all available interfaces on the servers, but there are some situations where the server

may prefer to be bound only to specific IP addresses. For example, a server running on a smartphone might want to be bound to the IP address of the WiFi interface but not on the 3G interface that is more expensive.

- `socket.recvfrom` is used to receive data from the underlying networking stack. This method returns both the sender's address and the received data.

The code below illustrates a very simple server running above the connectionless transport service that simply prints on the standard output all the received messages. This server uses the TCP/IPv6 networking stack.

```
import socket, sys

PORT=int(sys.argv[1])
BUFF_LEN=8192

s=socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
s.bind(('',PORT,0,0))
while True:
    data, addr = s.recvfrom( BUFF_LEN )
    if data=="STOP" :
        print "Stopping server"
        sys.exit(0)
    print "received from ", addr, " message:", data
```

A server that uses the reliable byte stream service can also be built above the socket API. Such a server starts by creating a socket that is bound to the port that has been chosen for the server. Then the server calls the `socket.listen` method. This informs the underlying networking stack of the number of transport connection attempts that can be queued in the underlying networking stack waiting to be accepted and processed by the server. The server typically has a thread waiting on the `socket.accept` method. This method returns as soon as a connection attempt is received by the underlying stack. It returns a socket that is bound to the established connection and the address of the remote host. With these methods, it is possible to write a very simple web server that always returns a *404* error to all *GET* requests and a *501* errors to all other requests.

```
# An extremely simple HTTP server

import socket, sys, time

# Server runs on all IP addresses by default
HOST=''
# 8080 can be used without root priviledges
PORT=8080
BUFLEN=8192 # buffer size

s = socket.socket(socket.AF_INET6, socket.SOCK_STREAM)
try:
    print "Starting HTTP server on port ", PORT
    s.bind((HOST,PORT,0,0))
except socket.error :
    print "Cannot bind to port :",PORT
    sys.exit(-1)

s.listen(10) # maximum 10 queued connections

while True:
    # a real server would be multithreaded and would catch exceptions
    conn, addr = s.accept()
    print "Connection from ", addr
    data=''
    while not '\n' in data :  # wait until first line has been received
        data = data+conn.recv(BUFLEN)
    if data.startswith('GET'):
        # GET request
        conn.send('HTTP/1.0 404 Not Found\r\n')
        # a real server should serve files
    else:
```