ignore the IPv4 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.

3.3 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.

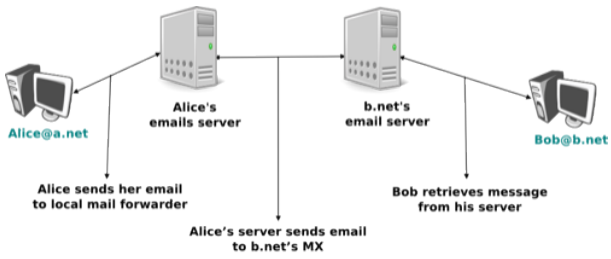

Fig. 3.6: Simplified architecture of the Internet email

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.

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.

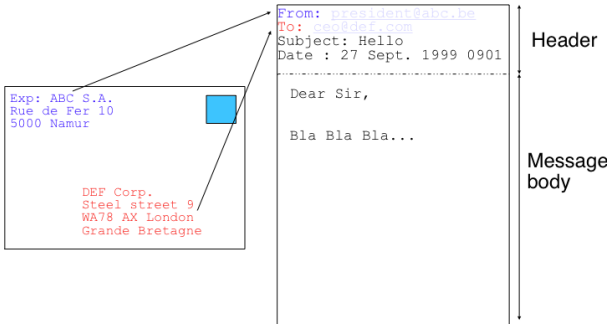

Fig. 3.7: The structure of email messages

- 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 [2] . 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.

```
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 [1]. 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.

[2] 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.

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

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>

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 once when the last group contains two bytes and twice 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 numerical preference is used first **RFC 2821**. 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*).

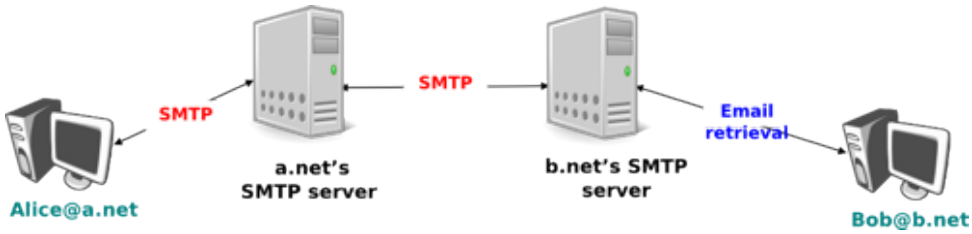

Fig. 3.8: Email delivery protocols

3.3.1 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 [3], 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.

[3] 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.

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
```

Fig. 3.9: 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* [4]. *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
```

Fig. 3.10: BNF specification of the SMTP responses

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

[4] 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**.

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 [5]. 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

[5] 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.

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** .

3.3.2 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 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).

the MAC key over the concatenation of the message counter and the unencrypted message. The message counter is not transmitted, but the recipient can easily recover its value. The `MAC` is computed as $mac = MAC(key, sequence_number||unencrypted_message)$ where the key is the negotiated authentication key.

Note: Authenticating messages with HMAC

ssh is one example of a protocol that uses Message Authentication Codes (MAC) to authenticates the messages that are sent. A naïve implementation of such a MAC would be to simply use a hash function like SHA-1. However, such a construction would not be safe from a security viewpoint. Internet protocols usually rely on the HMAC construction defined in **RFC 2104**. It works with any hash function (*H*) and a key (*K*). As an example, let us consider HMAC with the SHA-1 hash function. SHA-1 uses 20 bytes blocks and the block size will play an important role in the operation of HMAC. We first require the key to as long as the block size. Since this key is the output of the key generation algorithm, this is one parameter of this algorithm.

HMAC uses two padding strings : *ipad* (resp. *opad*) which is a string containing 20 times byte `0x36` (resp. byte `0x5C`). The HMAC is then computed as $H[K \oplus opad, H(K \oplus ipad, data)]$ where $\oplus$ denotes the bitwise XOR operation. This computation has been shown to be stronger than the naïve $H(K, data)$ against some types of cryptographic attacks.

Among the various features of the `ssh` protocol, it is interesting to mention how users are authenticated by the server. The `ssh` protocol supports the classical username/password authentication (but both the username and the password are transmitted over the secure encrypted channel). In addition, `ssh` supports two authentication mechanisms that rely on public keys. To use the first one, each user needs to generate his/her own public/private key pair and store the public key on the server. To be authenticated, the user needs to sign a message containing his/her public key by using his/her private key. The server can easily verify the validity of the signature since it already knows the user's public key. The second authentication scheme is designed for hosts that trust each other. Each host has a public/private key pair and stores the public keys of the other hosts that it trusts. This is typically used in environments such as university labs where each user could access any of the available computers. If Alice has logged on `computer1` and wants to execute a command on `computer2`, she can create an `ssh` session on this computer and type (again) her password. With the host-based authentication scheme, `computer1` signs a message with its private key to confirm that it has already authenticated Alice. `computer2` would then accept Alice's session without asking her credentials.

The `ssh` protocol includes other features that are beyond the scope of this book. Additional details may be found in *[BS2005]*.

3.5 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.

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 chose 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.

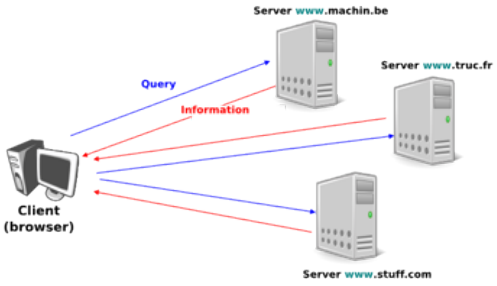

Fig. 3.11: World-wide web clients and servers

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 [1]. The most frequent scheme is *http* that will be described later. A URI scheme can be defined for almost any application layer protocol [2]. 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.
↪BaseHTTPRequestHandler
telnet://[2001:db8:3080:3::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:db8:3080:3::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* 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.

[1] 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 made 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.

[2] The list of standard URI schemes is maintained by IANA at http://www.iana.org/assignments/uri-schemes.html

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.

Fig. 3.12: 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 Transfert 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

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

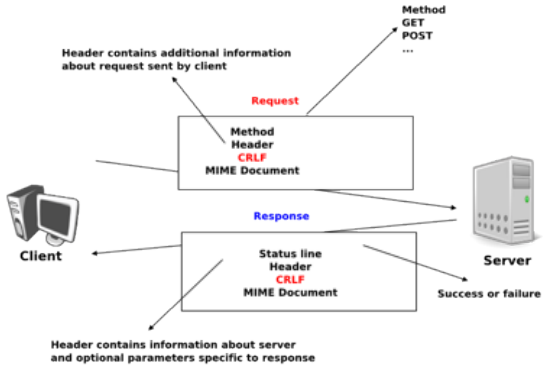

Fig. 3.13: HTTP requests and responses

- 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.

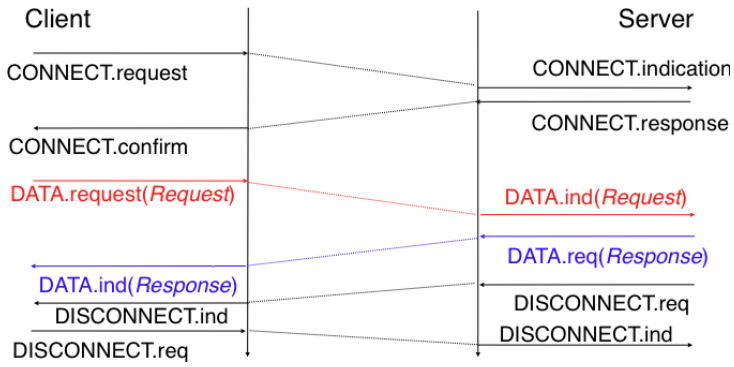

Fig. 3.14: HTTP 1.0 and the underlying TCP connection

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.

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

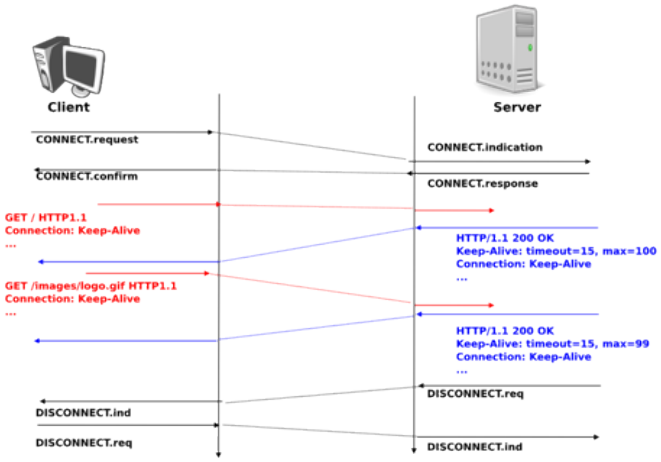

Fig. 3.15: HTTP 1.1 persistent connections

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 [3] , 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 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

[3] 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.

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.

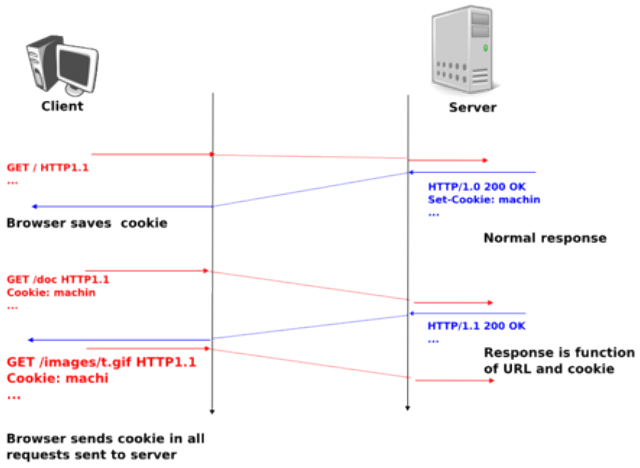

Fig. 3.16: HTTP cookies

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

3.6 Remote Procedure Calls

In the previous sections, we have described several protocols that enable humans to exchange messages and access to remote documents. This is not the only usage of computer networks and in many situations applications use the network to exchange information with other applications. When an application needs to perform a large computation on a host, it can sometimes be useful to request computations from other hosts. Many distributed systems have been built by distributing applications on different hosts and using *Remote Procedure Calls* as a basic building block.

In traditional programming languages, *procedure calls* allow programmers to better structure their code. Each procedure is identified by a name, a return type and a set of parameters. When a procedure is called, the current flow of program execution is diverted to execute the procedure. This procedure uses the provided parameters to perform its computation and returns one or more values. This programming model was designed with a single host in mind. In a nutshell, most programming languages support it as follows :

1. The caller places the values of the parameters at a location (register, stack, ...) where the callee can access them