Application layer

This is the layer at which users (including the operating system) operate.

There are many tools to use and many protocols that are used by these tools:

Tool	Protocol
browsers	http, HTML
mailers	SMTP
ftp	ftp
telnet	telnet
my own	my own

Each protocol has two parts: the client side and the server side. Sometimes, the same process acts both as a server and as a client at the same time (e–mail servers).

Quality of service

Applications serve a purpose and therefore must perform according to some standard of quality. The basic aspects of QofS are:

- Data loss.
- Time delay and jitter.
- Error rate in lossless protocols.
- Behind the scene, the cause of all those is the bandwidth available (and how it can vary).

WWW

The **World Wide Web** is a large–scale collection of *pages* that can be accessed by anyone. Important concepts:

- **Page** (or *document*) which can be an image or a file of any type.
- **Address** of a page, which is called *Universal Resource Locator* or URL. The format of a URL is:

  protocol://computer_name:**port**/*document_name*

  The **:port** part is optional (servers have port numbers assigned to them and most are standard numbers) and seldom used. Many protocols are in use (*file, ftp, http*).
- **Browser** which is a client process that makes access to WWW easy.
- **Web server** is a server process that is willing to accept WWW requests.

Web servers

Typical WWW servers use **http** as a communication language/protocol. However, it is not mandatory to use http. Anyone can write a browser or a Web server. They don't have to use http as long as both sides (browser and server) obey a common protocol.

A legal http server uses TCP/UDP **port** 80 (a transport layer concept). If you want to implement your own server, you have to occupy a different port or remove the http server from your system (if there is one).

http

See RFC2616 for details.

The term **http** denotes both a protocol and a specific command language which also is called http. The http language is designed specifically for client–server communication and is made of two distinct parts: **requests** and **responses**.

http clients set up sessions by contacting an http server (some browsers show the message *Contacting ...* followed by *Connect:* etc.). The http standard requires servers to use TCP and the **Socket** API; hence an http client must send requests using TCP.

http operates in two modes:

Nonpersistent mode where a new connection is open for each request separately.

Persistent mode where a connection is sustained until TCP times out or the connection is closed by the server or the client.

The current standard of http (1.1) uses persistent mode as default although many 1.1 servers close the connection after each response.

Request and response messages

http messages are in plain text (ASCII) so they are compatible with any system.

A request (sent by a client) has a **name** such as **PUT**, **POST**, etc.

A response has a **number code** which is followed by a verbose explanation, such as **200 OK** or **400 Bad request**, etc.

Format of http messages

http messages have a variable format and are divided into 3 parts:

Request/Status line:

```
GET /path/file.html HTTP/1.1
```

or

```
HTTP/1.1 400 Bad request
```

This is a single line terminated by a **CR/LF** sequence (in C–like languages `\r\n`).

Header lines having a format:

Fieldname: value CR/LF

Each line ends with a **CR/LF** (`\r\n`).

The header is terminated by an empty line, which is a **nothing** placed between two end–of–line sequences: **CR/LF/CR/LF** (`\r\n\r\n`) (spot the nothing in the middle).

Fieldnames are case–insensitive.

Body: empty or any text.

Message body

The body is empty or its presence is signalled by the `Content-Length` field.

If present, the message body is made of “arguments” for a search request or the requested text of a response. The type of the content may be identified by a `Content-Type` header line.

A response may contain a body even without a `Content-Length` field present if (and only if) it is followed by a connection tear–down, i.e. a **close()**.

A sample request

```
GET /somedir/page.html HTTP/1.1
Host: www.someplace.edu
Connection: close
User-agent: Mozilla/5.0 (...details...) ... Firefox 3.5.3
```

User-Agent field

This field can contain any text and may even be misleading. If correct, it gives:

- The name of the browser software supported.
- (In parentheses) Operating system information and encryption standard, as in:

  `(Windows; U; Windows NT 5.1; rv:1.9.1.3)`

  Here, the "U" stands for USA. Other possibilities are "N" (none) and "I" (international); both are obsolete.
- The layout engine (**Gecko** for Mozilla–based browsers, or **Trident** for MSIE; there is also **WebKit** for Safari, etc.). The engine's version may show in the parenthesized part described above, as in `rv:1.9.1.3` which is Gecko's August/2009 version.
- Spoofing browsers/servers sometimes append to the line the name of the real browser.

The `User-agent` field is a favourite place for **spoofing**: MSIE often pretends to be **Mozilla**; Servers running other packages sometimes go as far as pretending to be simultaneously “Mozilla”, “MSIE” and, say, “Opera” in order to capture as much traffic as possible.

On a different front, the user–agent field is used for **sniffing** which gives a way to display contents in a browser–oriented manner. This can be done for nefarious purposes, as in MSIE (allowing non–standard functionality for displaying text) or for user–friendliness, as in mobile phones.

Try header viewer to see what various servers send to your site when approached. If you want to see the header of your own browser’s http messages, try my own header.

Another request

```
GET /somedir/page.html HTTP/1.1
Host: www.someplace.edu
Keep-Alive: timeout=90
Connection: Keep-Alive
User-agent: Mozilla/5.0
Accept-language: en-US
```

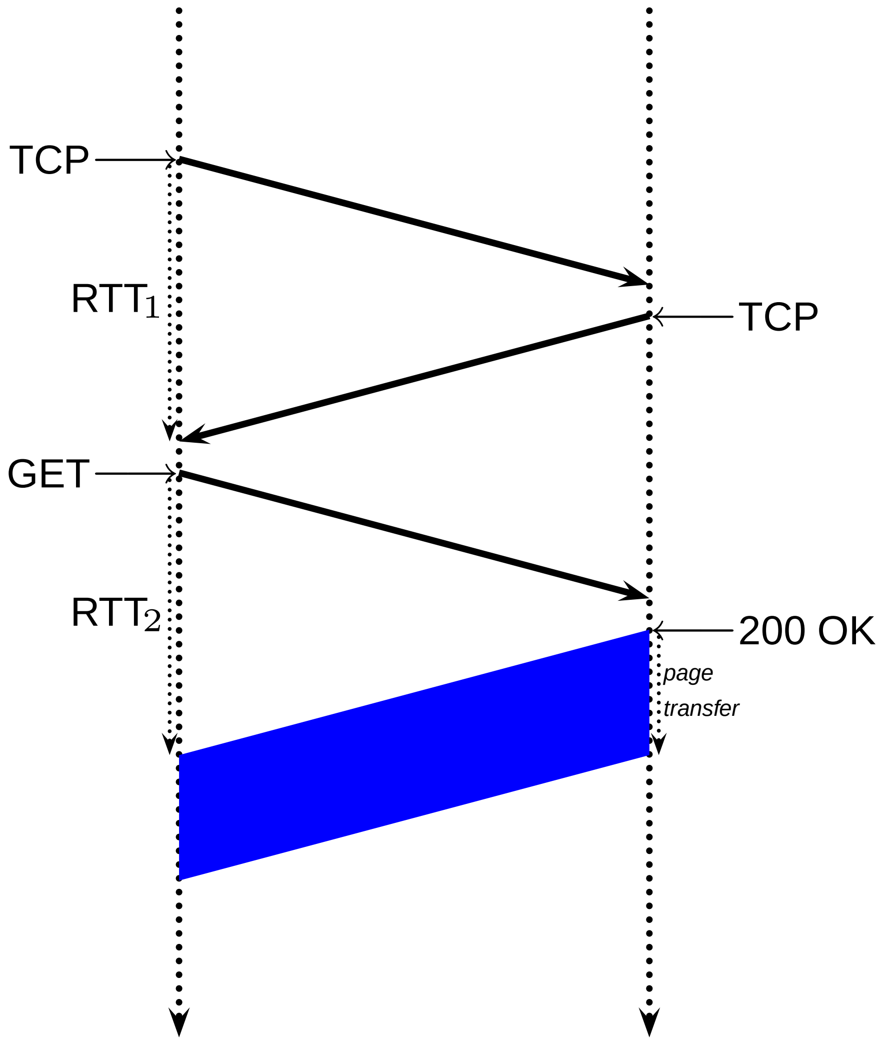
A simple response

```
HTTP/1.1 200 OK
Date: Fri, 15 Aug 2008 16:56:17 GMT
Server: Apache/2.2.9
Content-Length: 1258
Keep-Alive: timeout=90
Connection: Keep-Alive
Content-Type: text/html

```

⟵There is an empty line here

The empty line is (in this case) followed by a message body 1258 bytes long.


TCP
RTT$_1$
TCP
GET
RTT$_2$
200 OK
page
transfer

Persistent connections

http 1.1 requires all connections to be **persistent**, i.e. connections, once established successfully, will not be closed by either party without prior warning.

This is impossible to implement without asking for a DoS attack hence the rules are more detailed:

- Each party may demand that the connection be closed by placing in the header the field:

  ```
  Connection:  close\r\n
  ```

- A node may close a connection without warning if a predetermined **timeout** occurs.

Timeouts are exchanged between the client and the server by the use of the header field `Keep-Alive`:

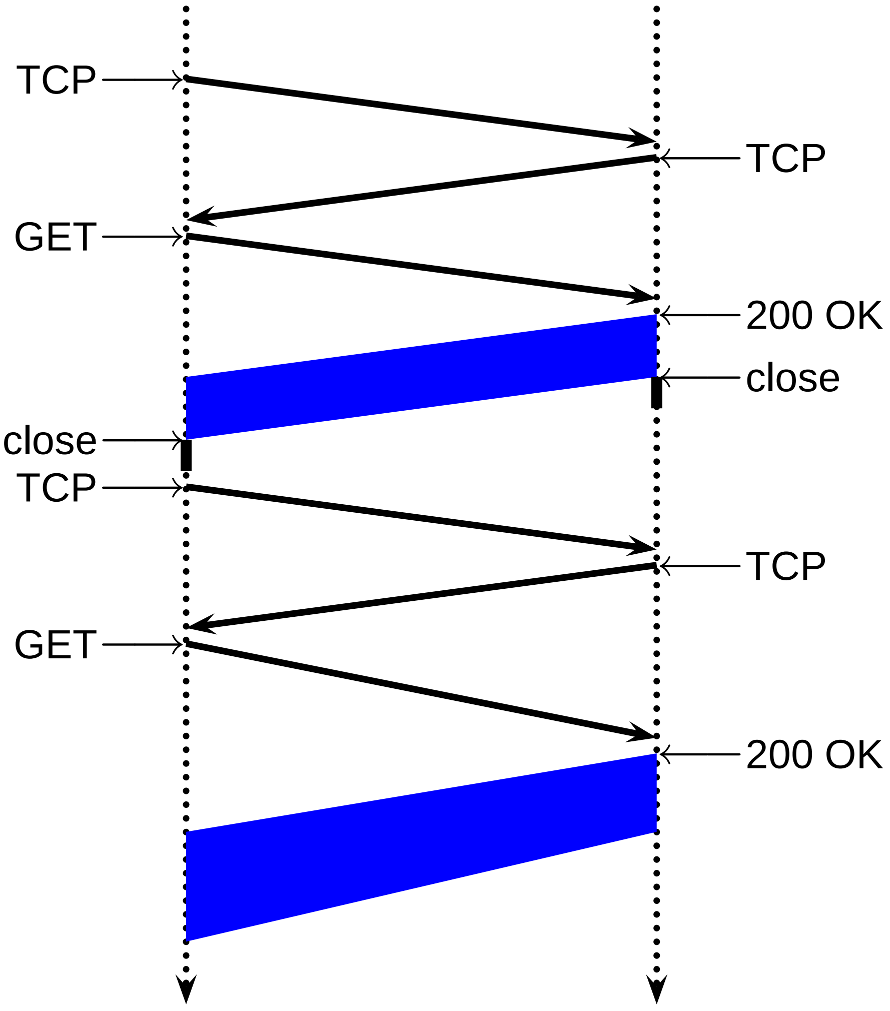
```
Keep-Alive:  timeout=90\r\n
Keep-Alive:  90\r\n
```

or the field:

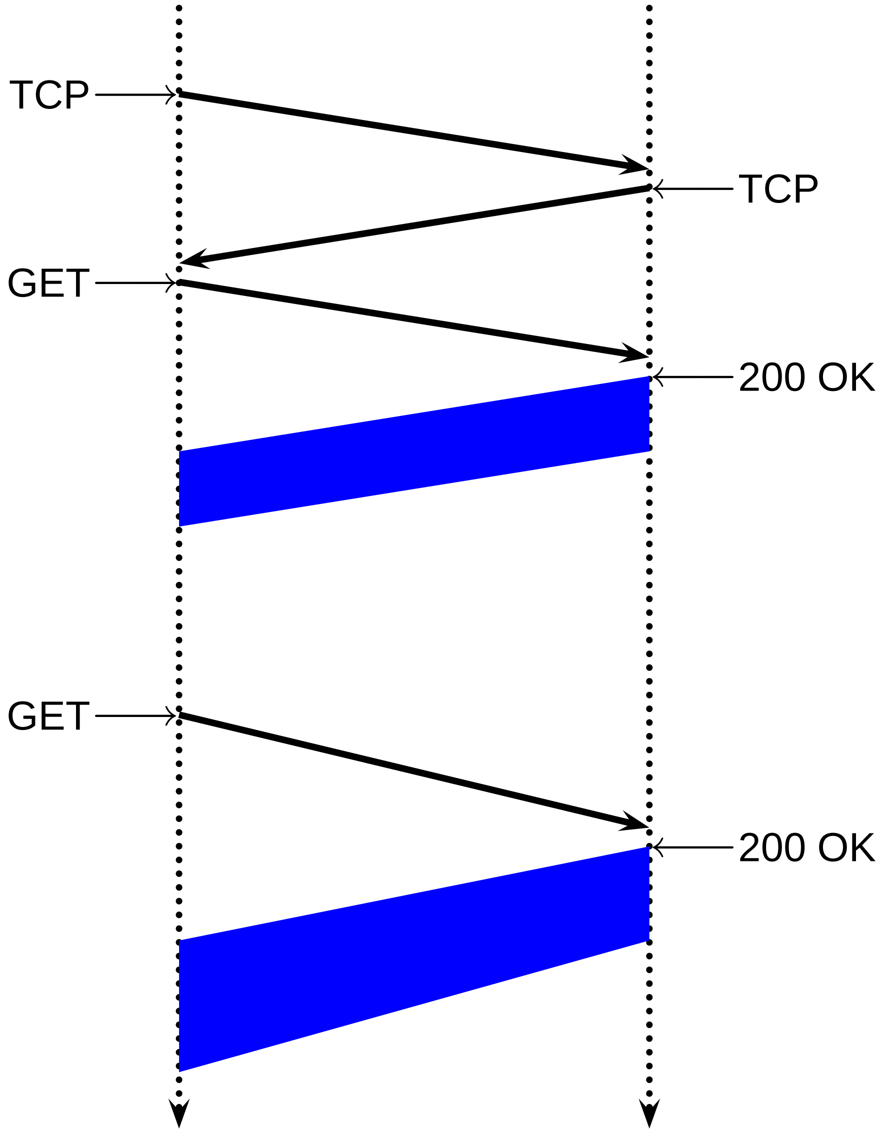
```
Connection:  Keep-Alive\r\n
```

which is almost meaningless in 1.1; its main purpose is to prevent the other side’s timeout from occurring. Beware of numerous variations of the interpretation of `Keep-Alive`. For example, Apache does it in its own way.

Non–persistent “session”



A persistent session



Proxies

A user may choose to declare a **proxy** for satisfying http **GET** requests.

A proxy is a server that mirrors some of the contents of a real http server by caching pages passing through it.

When a client issues a **GET** request, the proxy checks whether the requested page is in its cache.

- If not, it sends a **GET** to the real server and forwards the server's response to the client.
- If it has a copy of the page, it sends a conditional request to the real server, asking for the page only if it was modified since it was cached. Then it takes suitable action.

FTP

The File Transfer Protocol (**RFC959** and RFC 2228) is an old application that performs the same task as http. The application is made of 3 pieces of software:

- **FTP** user interface.
- **FTP** client residing on the user's computer.
- **FTP** server residing on a remote computer.

Like http, **FTP** uses **TCP**, but it establishes 2 connections from the client to the server.

FTP connections

Control connection (port 21) is used to pass requests and responses between the FTP client and the FTP server.

Data connection (used to be port 20) is used to transfer files between the local and remote file systems.

A protocol that uses a separate connection for control packets is said to pass these packets **out–of–band** (e.g. **FTP**). Otherwise, it passes them **in–band** (e.g. http).

Connection modes

An **FTP** server is always listening (Socket API: socket(), bind(), listen()) for incoming connection requests on its **control port**; IANA assigned port 21 for this purpose.

A session is established between a client and a server through the following sequence of actions:

- A client starts an **FTP** session by connecting to the server's **FTP** port (21). (Socket API: socket(), connect()).
- The server accepts the connection (Socket API: accept()). This sets up a **control connection** circuit.
- The client starts establishing a **data connection** circuit. There are two ways of doing this: a **passive** or an **active** mode.
- The server and the client finalise building the data circuit and are ready for an exchange of files.

Passive data connection

- The client sends to the server (say, 131.104.48.133:21) a request packet consisting of the keyword only: **PASV**
- The server replies (through its port 21):

227 Entering Passive Mode (IP–address,port)

The pair IP–port may look like this:

227 Entering Passive Mode (131,104,48,133,249,133)

which reads: IP address 131.104.48.133 and port number 63877 ($249 \times 2^8 + 133 = 63877$).

- The client creates a socket and connects to port 63877 of the node 131.104.48.133. Note that the server has the option of redirecting the service by giving someone else's IP address in response 227 (it seldom works).
- The server accepts the connection and the circuit is functional.

Active data connection

- The client opens a socket, chooses a port number, say 1394, and binds the socket to the port (1394).
- The client sends (to port 21) a packet containing the IP–port pair:

PORT **(131.104.49.234,5,114)**

where 131.104.49.234 is the client's IP address and $5 \times 2^8 + 114 = 1394$ is the port chosen by the client.

- The server opens a socket and connects it to 131.104.49.234:1394. Then it sends to the client a response:

200 PORT **command successful**

finishing the creation of the circuit.

SMTP

The **Simple Mail Transfer Protocol** is another commonly used network application (RFC 2821) dating from 1982.

SMTP is very similar to **FTP**. The differences are:

- It passes commands **in–band** (so it uses one TCP connection: port 25).
- Instead of copying files, **SMTP** appends messages to a predefined file called a **mailbox**.

The main difference between **SMTP** and http is that the former is a **push** service while the latter is a **pull** service (FTP has both services).