HTTP Primer

Contents

1. Introduction

1.1 Basic Model

In order for the Web to work, there was a need for a way of requesting Web pages and having them delivered from a remote server to the browser. In December 1990 when the first web pages were served from a server in CERN, the decision had already been made to use a very simple protocol called HyperText Transfer Protocol (HTTP) running over the Internet using TCP/IP. Appendix B gives an early paper indicating some of the design considerations. Routing the pages through the network and uniquely addreessing the destination is done by the Internet Protocol (IP). Communication between the hosts is achieved by the Transmission Control Protocol (TCP).

The HTTP architecture agreed is shown in Figure 1.1. Only the Client is allowed to make requests. It asks the server on the Internet at the site www.w3.org to deliver the page file.htm to the Client for browsing. The Server may be able to find the file (called a document or a page) and, if so, it will deliver it to the Client. If the file cannot be located, an error response is sent instead.

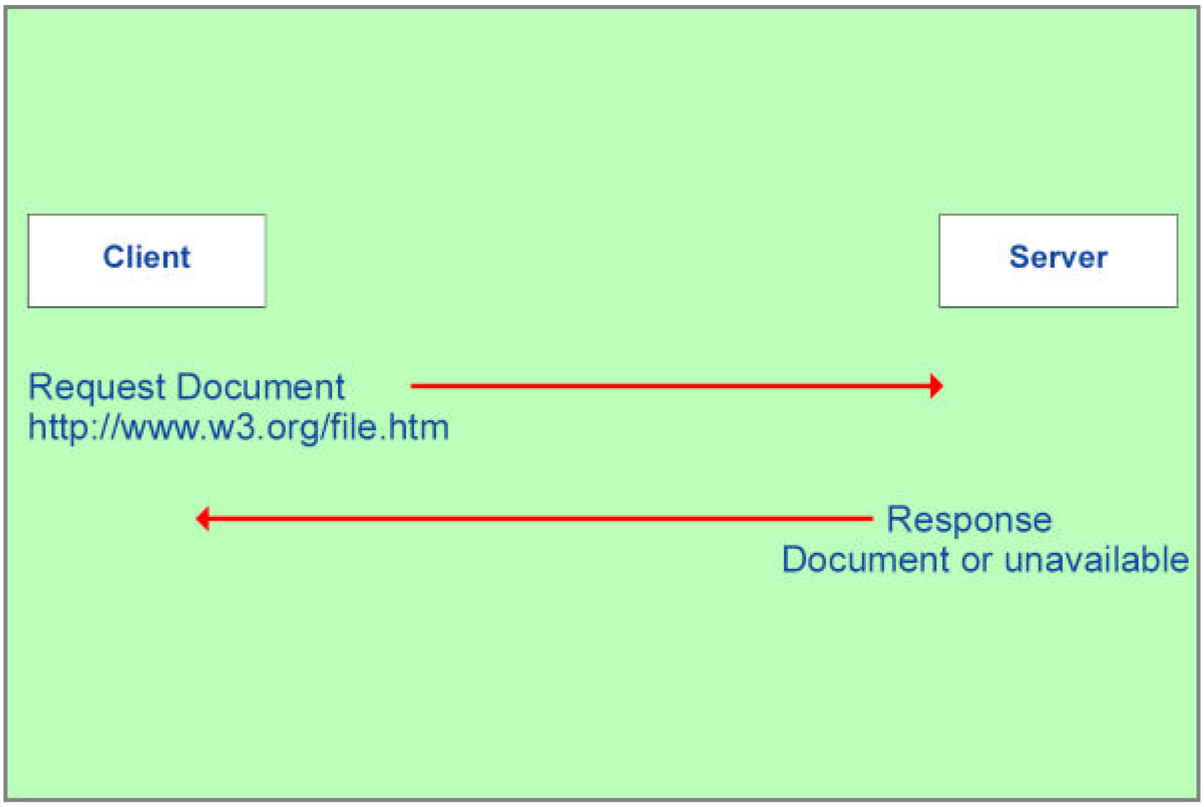


Figure 1.1: Request/response Model of HTTP

The decision was made for HTTP to use TCP/IP as the underlying addressing and transportation system. IP delivers packets over the Internet based on the IP addressing scheme (more of that later) which consists of a 32-bit number expressed as four bytes with the decimal value of each byte given (something like 127.23.56.96 where the four numbers are all less than 256). The mapping from the symbolic name (www.w3.org) into the 32-bit IP address is achieved through a DNS Server (see Figure 1.2).

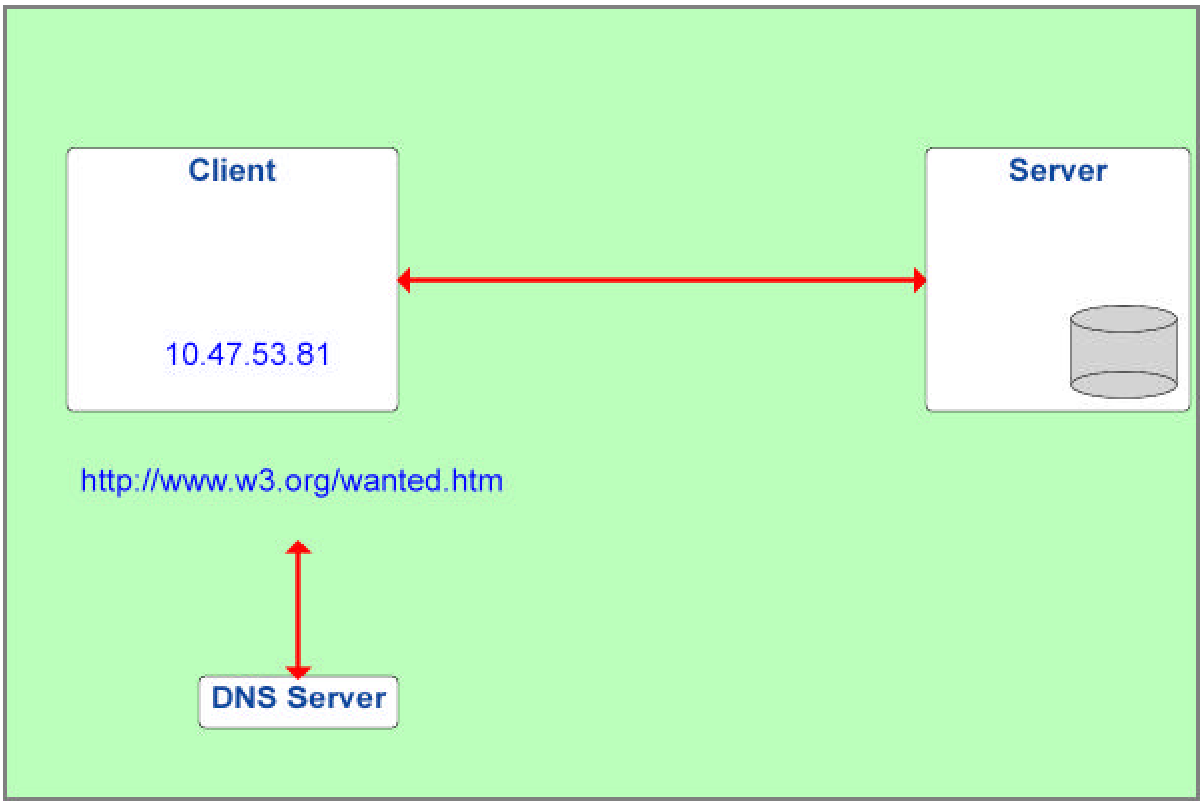


Figure 1.2: Domain Name Server

1.2 Caching and Proxies

As soon as the web started getting used, it became clear that this simple model was not that efficient. Suppose the user downloads a web page and then a second. If the user wants to refer back to the first page, then the model as it stands would require the page to be requested again. In consequence, almost from day one the possibility of having a **cache** of already downloaded web pages was seen as useful. When the user hits the **back button** on the browser, the page is retrieved from a local cache on the computer running the browser rather than request the page again. Browsers have over the years added other features such as History files and Bookmark lists which also can make use of a local cache.

At the Server end, there is also a use for a cache. If the server has a large main memory, rather than retrieve each file from disc, it is able to keep the last set of pages requested in main memory in order to improve its response to users.

Organisations soon began to find that their staff were using the Web a great deal. In some cases this was good news as they were accessing relevant information needed to do their job. Sometimes the staff were downloading information of a more private nature, organising their holidays, purchasing goods etc. This led to the introduction of a device called a **proxy** which to the Client looks like a Server and to a Server looks like a Client. Proxies have a wide range of uses. They can be used to control what pages the organisation's staff are allowed to access. If the proxy is placed on the **firewall** between the company's **Intranet** and the Internet, it is possible to check what documents are being requested. The organisation can set up the network so that all external requests go the proxy which onward routes the ones that the organisation is happy with and sends back an unavailable error message otherwise.

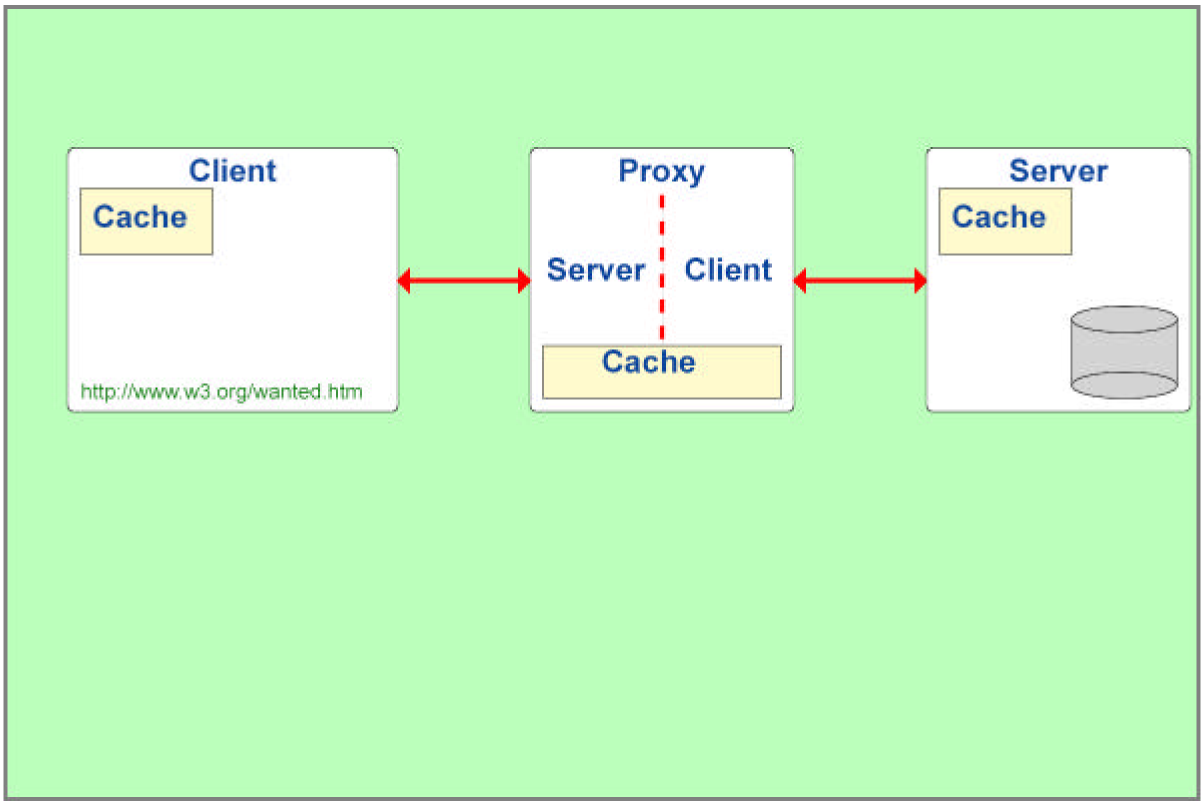


Figure 1.3: Caching and Proxies

Proxies also can contain caches. In this case, the cache serves a set of users rather than a single user. In an organisation, it is likely that a set of users will request similar pages due to the nature of their work, so a proxy cache also has value. User A requests a page that is stored in the proxy cache and when user B requests the same page, it is returned from the proxy rather than requesting it from the Server. Figure 1.3 shows the overall arrangement of a site with a proxy cache.

If a proxy is of use to an organisation, it may also be of value to a Department, to an Internet service Provider, to a Community and even a country. For example, the UK has an academic cache aimed at reducing the traffic between the UK and the USA to lower the need for expensive bandwidth between the UK and the USA. Some countries have national caches to censor information that arrives in the country.

Information cannot be left in a cache forever as eventually it will be out-of-date and the cache will fill up. The Server may have updated a Web page but all the caches are still returning the old version. With the potential of there being a whole series of caches between the user and the site from which a page is required, mechanisms are needed to handle **stale** pages in the various caches and to ensure that pages of the appropriate freshness are returned to the user. If the request is for a stock market price, the page **must** be fresh. If it is for the local weather forecast, a delay of an hour may be OK. If it is to discover the length of a metre in inches, a much longer period elapses before the page becomes invalid! It is only recently that the problems of stale information have been largely cured on the Web by changes to HTTP.

1.3 Plug-ins and Helpers

With the arrival of the Mosaic browser, it became quite easy to embed graphical images in Web pages. This has several repercussions on the simple request-response model. The Web page when it arrives is found to have embedded images and these also need to be retrieved. The load on the Internet increases and the images may need to be downloaded from different Web sites. Figure 1.4 shows the added complexity.

On the network side, the single page download results in several requests and responses. On the Client side, the browser either needs to be able to interpret the image arriving or upgrade itself to achieve this. The simplest method of doing this is to add a **plug-in** to the Client that is capable of rendering the image downloaded. The browser allocates an area of the page and the plug-in is responsible for **painting the pixels** within that area. Even today, browsers accept some formats and not others and not all browsers can install the same plug-ins. This leads to another use for the proxy. For example, if the browsers used by an organisation do not support an image format like PNG but do support GIF, it is possible for the proxy to provide a **transformation** between the original format and the one that can be supported.

Some additions to the Web page are sufficiently complex that the plug-in approach may not be sensible. Examples are audio and video. The basic protocols designed for the Web are based on the assumption that either the document is returned or it is not. There is no real concept of an approximation. With streaming media it is feasible that some information can be lost and the algorithms used to download the information may be different depending on the bandwidth available. In such cases, the browser employs a **helper** application to handle a particular media type. the helper may download the file using a different communication protocol from HTTP; a protocol better suited to the characteristics of the media type.

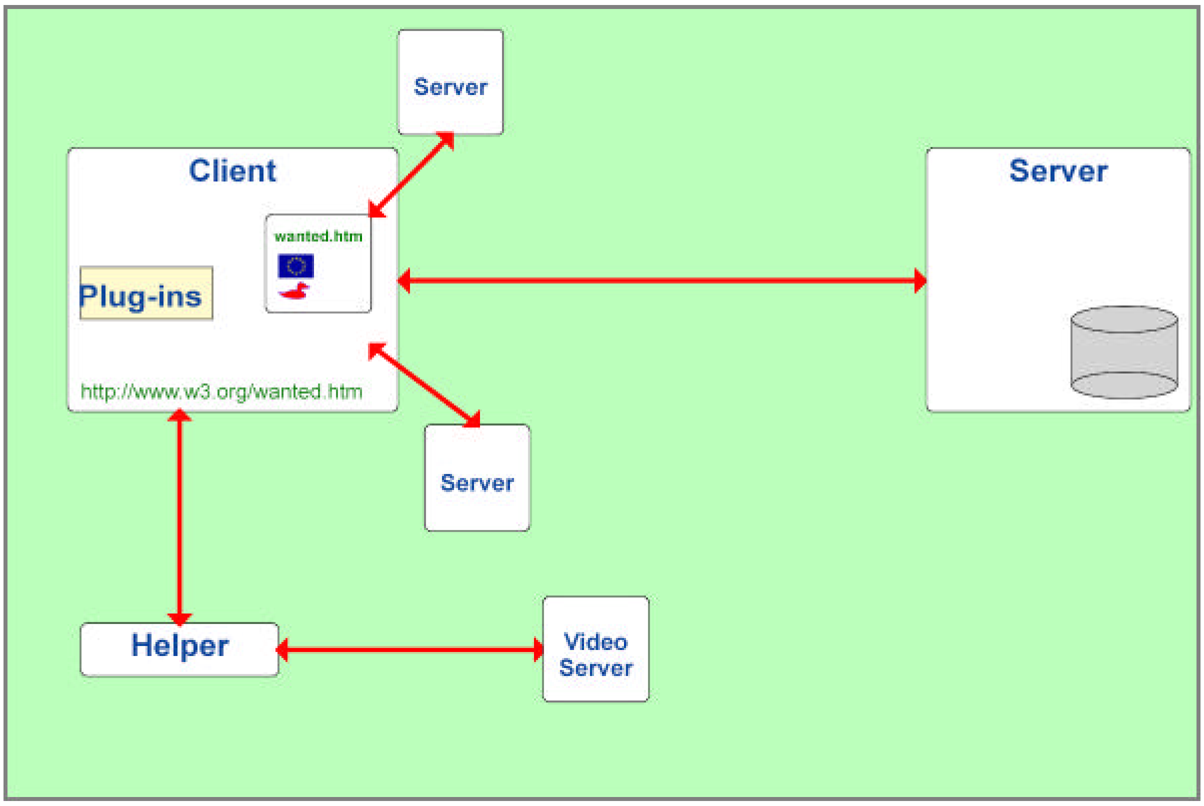


Figure 1.4: Images in Web Pages

1.4 Summary

So far we have looked at what initially appears to be quite a simple protocol requirement. The model is soon complicated by the reality of how the Web is used in the real world where the bandwidth on the Internet is limited and the requirements soon became more demanding.

In the history of the Web, we showed in Figure 1.5 the 7 layers of the ISO model and the equivalent Internet protocols, particularly TCP/IP. Before looking at HTTP further, we will first look at the basic infrastructure chosen to support HTTP, IP and TCP.

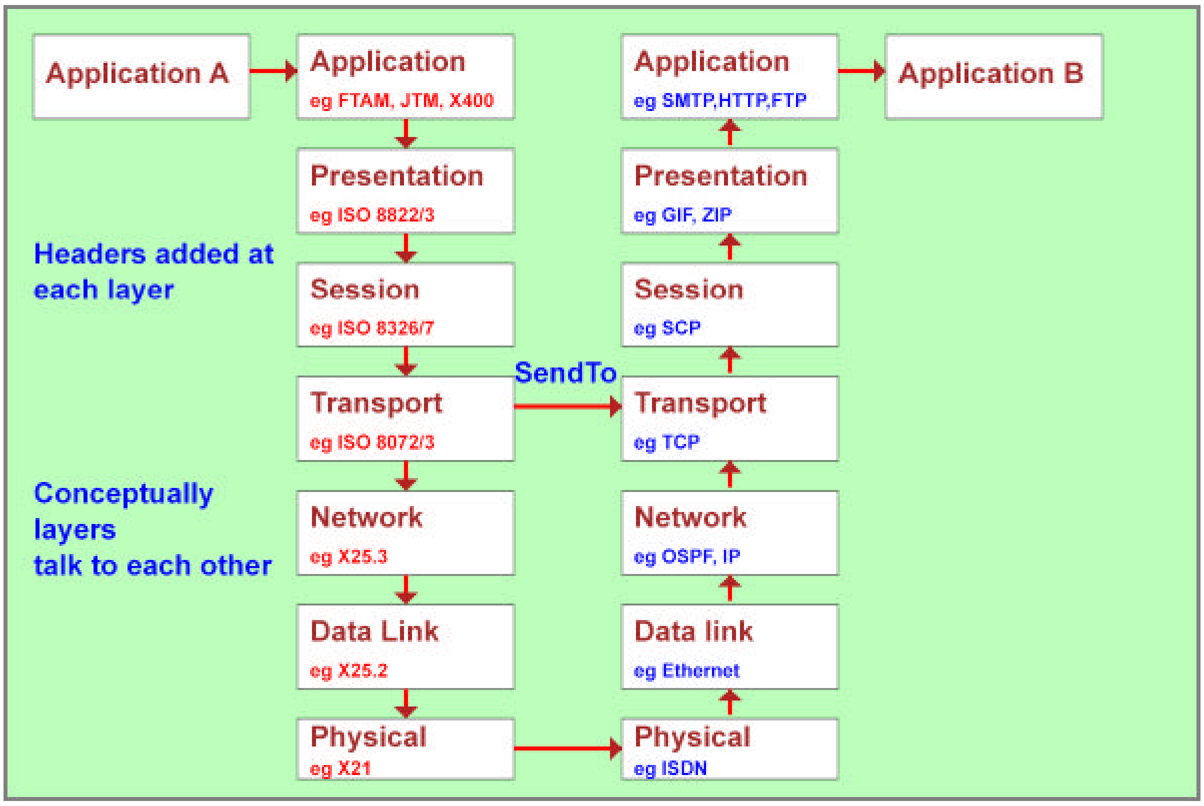


Figure 1.5: ISO 7-Layer Model

2. Internet Protocol

2.1 Introduction

As we have already seen, the Internet is a network of networks. The problem IP is addressing is how to achieve end-to-end communication across a potentially heterogeneous collection of intermediate networks along the way. The Internet Protocol (IP) is the network-level protocol that underlies the Internet. The main aim was to keep the Internet itself simple. IP defines an architecture based on sending individual **packets** from one host to another. Each host has a unique name, its **IP address**. A packet knows where it came from and where it is going. The **IP header** contains this information and it is followed by the data in the packet. The Internet consists of a set of routers that accept packets, look at their destination and decide where to route it next. No state is retained and it is feasible for the packet to arrive back at the same router. Routers make decisions for each packet based on its current understanding of the state of the Internet. Thus packets travel independently on different paths and they are likely to arrive at the host in a different order from when they left. Packets may not all follow the same route, and routes are not necessarily of the same length. Packets may be lost on the way or be corrupted when they arrive. If the Internet is congested, routers may need to store packages before forwarding them. If a route gets overloaded, it will throw packets away in order to relieve congestion.

By being so simple, IP can exist in a network environment that is overloaded and having transient failures. Clearly for many applications, this unreliable and unordered delivery is a problem, as it is for the Web. In consequence, sitting on top of IP is the Transmission Control Protocol (TCP) that has the problem of ensuring that packets are delivered reliably and in the right order. We will talk about TCP later.

2.2 IP Addressing

IP addresses are 32 bit numbers that can be considered as mailing addresses. Apart from some considerations that we do not need to bother about, the IP address refers to a specific machine on a defined site. In consequence, it was sensible to divide the address into two parts. The first part defines the place on the **network** where the specific machine sits and the second part, the **host** part, defines the machine at that location. In consequence, the routers on the Internet only need to know the network part of the address and can rely on the network at the site to deliver the package to the correct machine.

An organisation like Brookes will be allocated a set of IP addresses for its machines and, in a simple situation, a single network address. Unfortunately things are not that simple. How many bits should be allocated to the network part of the address and how many to the host part. Well that depends on the size of the site. As this varies, different Classes of sites were defined early on that correspond to large, medium and small sites. Figure 2.1 shows the Classes available:

Class A
: Really large sites of up to 16 million machines. 8 bits are used for the network address and 24 for the host.

Class B
: Medium size sites with up to 65000 machines. 16 bits are used for the network address and 16 for the host.

Class C
: Small organisations with up to 256 hosts. 24 bits are used for the network address and 8 bits are used for the host.

The addresses are written with full stops between each octet value written in decimal.

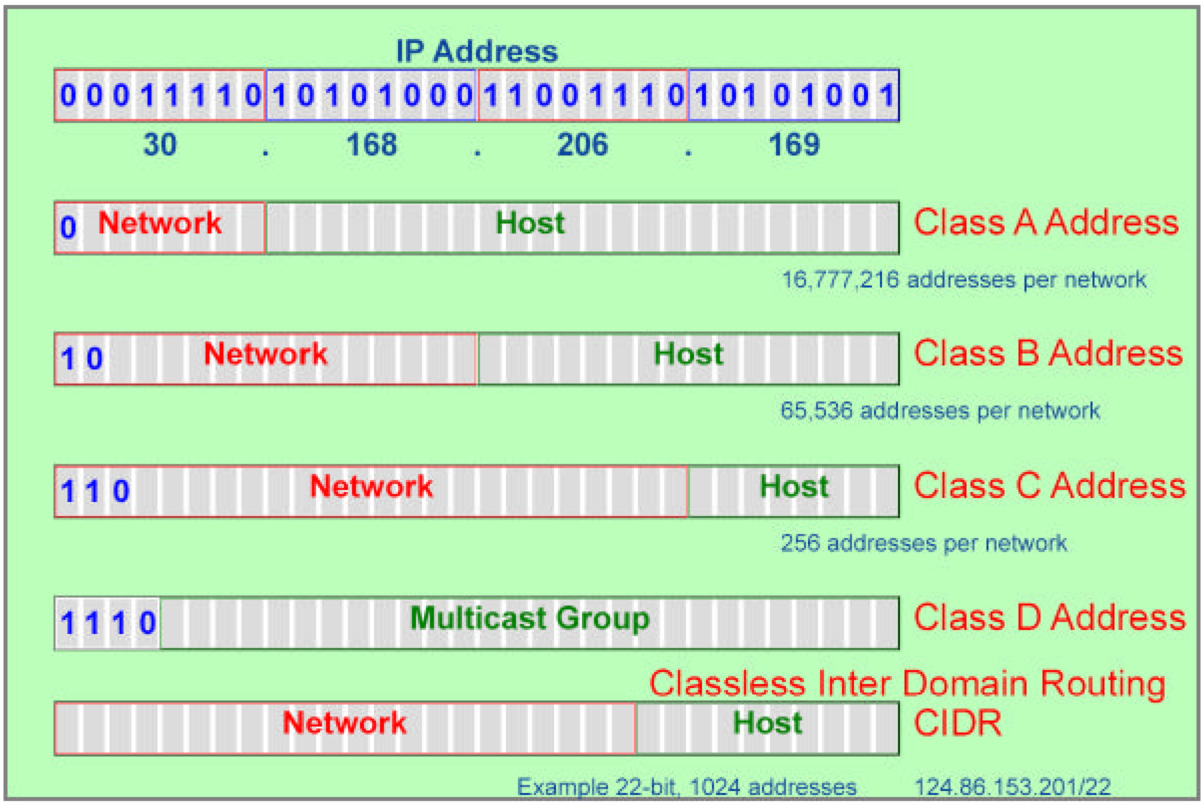

IP Address
0 0 0 1 1 1 1 0 1 0 1 0 1 0 0 0 1 1 0 0 1 1 1 0 1 0 1 0 1 0 0 1
30 . 168 . 206 . 169
0 Network Host Class A Address
16,777,216 addresses per network
1 0 Network Host Class B Address
65,536 addresses per network
1 1 0 Network Host Class C Address
256 addresses per network
1 1 1 0 Multicast Group Class D Address
Classless Inter Domain Routing CIDR
Network Host
Example 22-bit, 1024 addresses 124.86.153.201/22

Figure 2.1: IP Addresses

It does not take long to realise that this allocation of addresses will not work long term. Restricting the addresses to 32 bits and splitting them up in this way means that addresses will run out. Once everybody has their phone, watch, coffee machine, refrigerator, toothbrush and razor needing an IP address, there is not enough to go around. To slow down the problem, the restriction on fixed boundaries between the network and host parts was abandoned and Classless InterDomain Routing (CIDR) was introduced. This allows the split between the network and host part to be at any bit. Thus 124.86.153.201/22 defines an address where the top 22 bits represent the network address and the lower 10 represent the host address. It does not solve the problem but has meant it has gone away for a while.

2.3 IP Domain Name System

Host names on the Internet have the form www.w3.org and indicate more the type of organisation and where it lives rather than its size and when it was allocated a set of IP addresses. In consequence there is a need to translate from the host name part of the Universal Resource Locator (URL) to the IP address of that host. The key point is that the solution must be scaleable. A solution that works for a small number of IP addresses will not necessarily work as the number of connections grow. Figure 2.2 shows the hierarchical nature of the Domain Name System (DNS) architecture. There is a single root node and underneath this are a set of **top-level** domain names like com and org which define the type of the organisation and ones like uk and zw that relate to a country (zw stands for Zimbabwe). Recently some new generic top-level domain names have been introduced (aero, biz, coop, info, museum, name and pro). Underneath this top level are second-level domains like ac for the academic community in the UK. Finally below that are even more subclassifications. Thus brookes.ac.uk is a university in the academic sector of the UK.

Given the DNS name, to find the IP address requires somebody to have the translation from one to the other. rather than have a single machine that handles all names (that would not work especially if it broke!), each subtree in the DNS looks after its own translation and makes sure this carries on happening even if the primary translator fails.

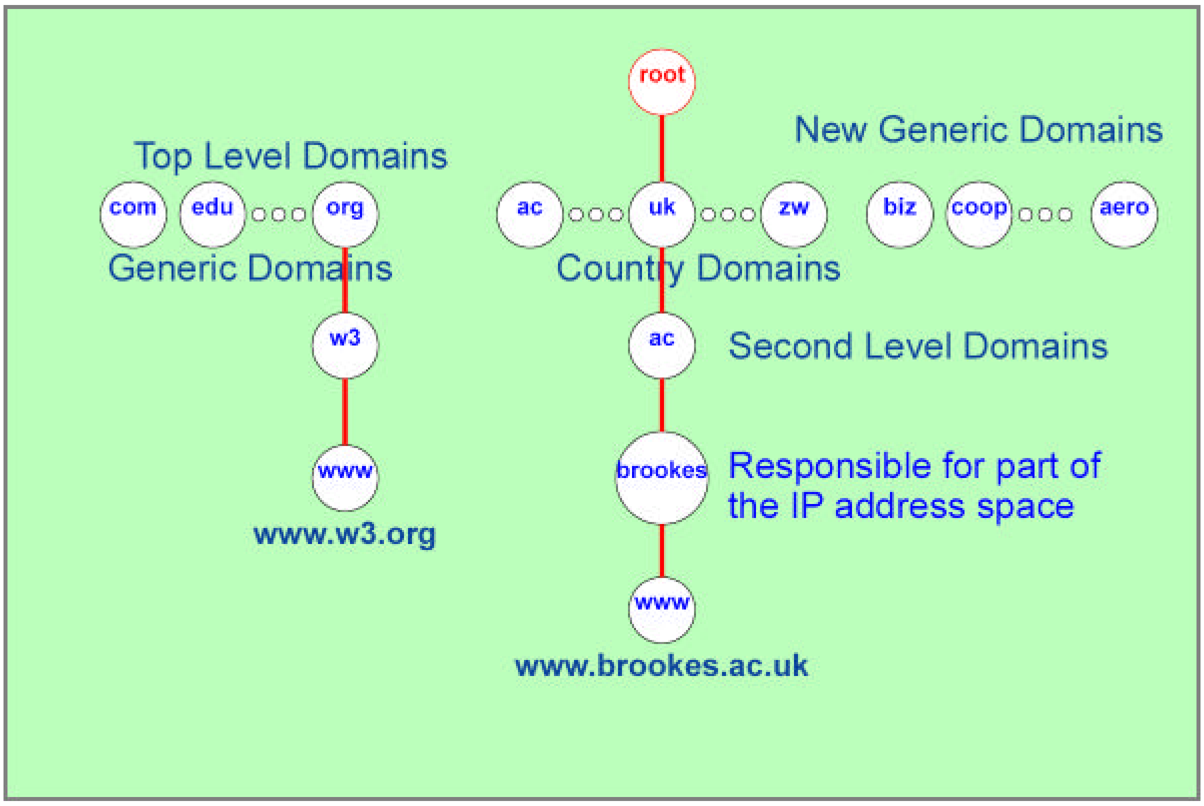


Figure 2.2: Domain Name System

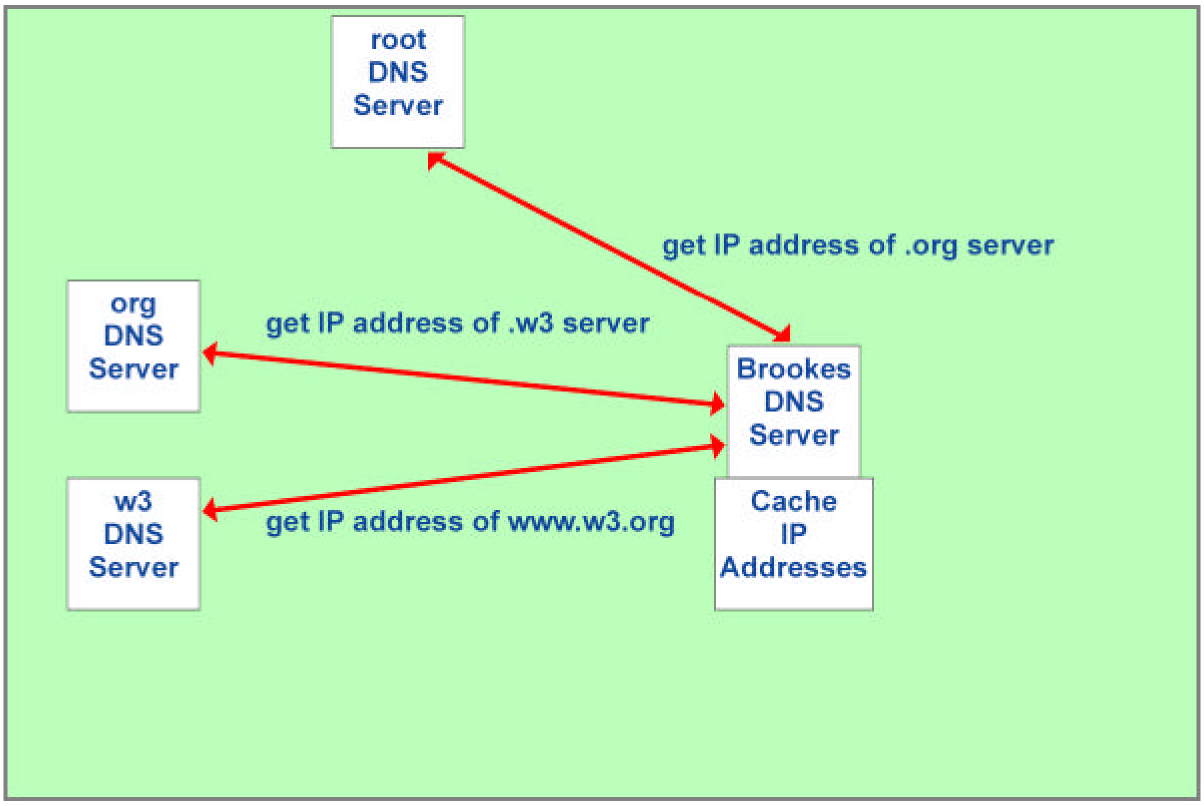

root DNS Server
get IP address of .org server
org DNS Server
get IP address of .w3 server
Brookes DNS Server
w3 DNS Server
get IP address of www.w3.org
Cache IP Addresses

Figure 2.3: Conversion to IP Address

Figure 2.3 shows how a browser at Brookes would find out the IP address of www.w3.org to request a page from there. In its simplest form, the local name server at Brookes would contact the root to find out where the name server for the **org** domain was. Having this information, it would ask it to give it the adress of w3 and so on. It is possible to ask one name server to onward route the request but the general idea is the same. The name servers tend to have caches of previous requests so that this rather long winded way of establishing the address is not always required.

Just as a protocol is needed to route HTTP requests, the same is true for domain name translations. These will tend to be simple. The packet sizes in each direction are small and one IP packet in each direction will do the trick. The User Datagram Protocol (UDP) sitting on top of IP is good enough for the job. This is an unreliable protocol. If you do not get a response, you need to make a judgement on how long to wait before making the request again. The header just consists of the source, destination, length and a checksum. There is no need to establish any long-term connection and it fits the bill for this activity. However it would not be any use for HTTP that needs a better expectation of longer transmissions being completed successfully. To do this it would need to repeat what other underlying protocols already provide.

2.4 IP Packets

Figure 2.4 shows the general structure of an IP packet. The data is preceded by a header that is normally 5 32-bit words (20 bytes) long. Most of the fields are self evident. The identification field contains a unique value defined by the sender. If a packet has to be split to get through the Internet, this unique identification is used to put it back together again. The fragment field is used to indicate which part of the split packet, individual sections are. The **Time To Live** field is important as it defines how many hops the packet will take before self disintegration. each router decrements the count by one. It provides a crude way of getting round loops in the system. When the IP packet reaches the other end, it may be part of several protocols. We have already talked about UDP and will talk about TCP. It is important to know where the packet is supposed to go when it reaches its destination.

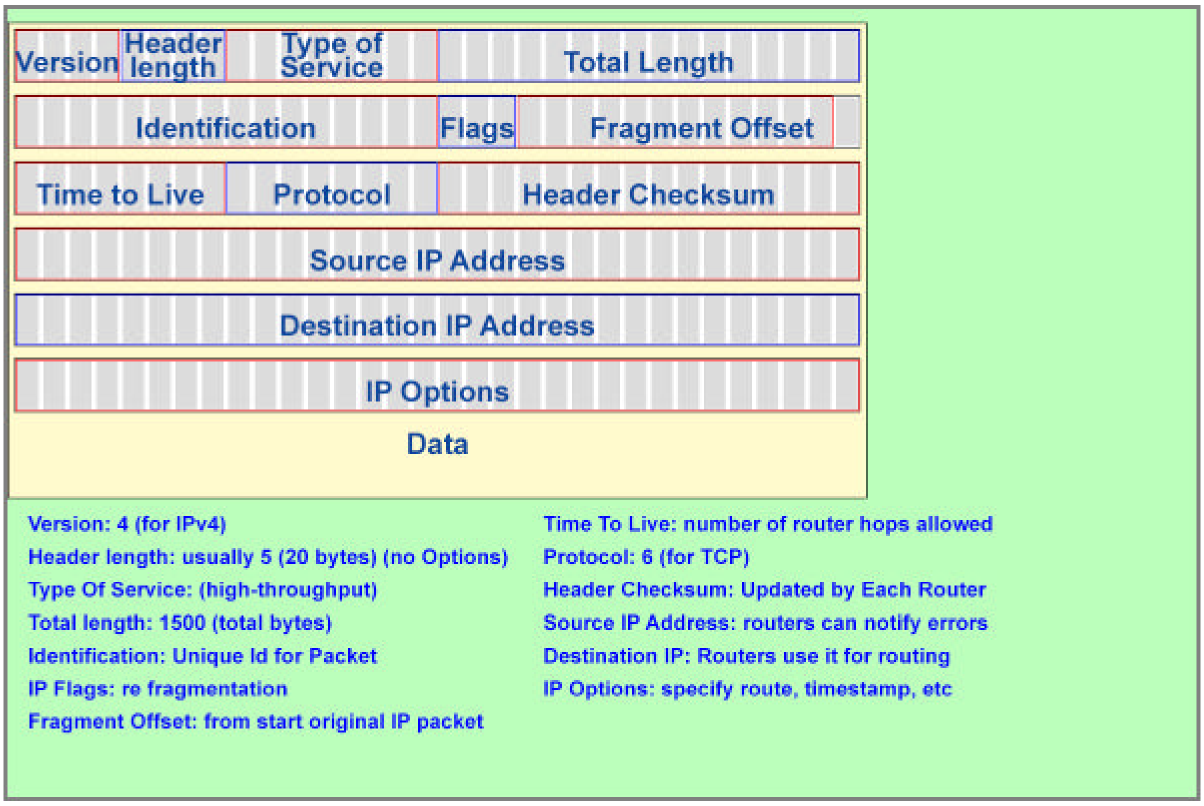

Version
Header length
Type of Service
Total Length
Identification
Flags
Fragment Offset
Time to Live
Protocol
Header Checksum
Source IP Address
Destination IP Address
IP Options
Data
Version: 4 (for IPv4)
Header length: usually 5 (20 bytes) (no Options)
Type Of Service: (high-throughput)
Total length: 1500 (total bytes)
Identification: Unique Id for Packet
IP Flags: re fragmentation
Fragment Offset: from start original IP packet
Time To Live: number of router hops allowed
Protocol: 6 (for TCP)
Header Checksum: Updated by Each Router
Source IP Address: routers can notify errors
Destination IP: Routers use it for routing
IP Options: specify route, timestamp, etc

Figure 2.4: IP Header

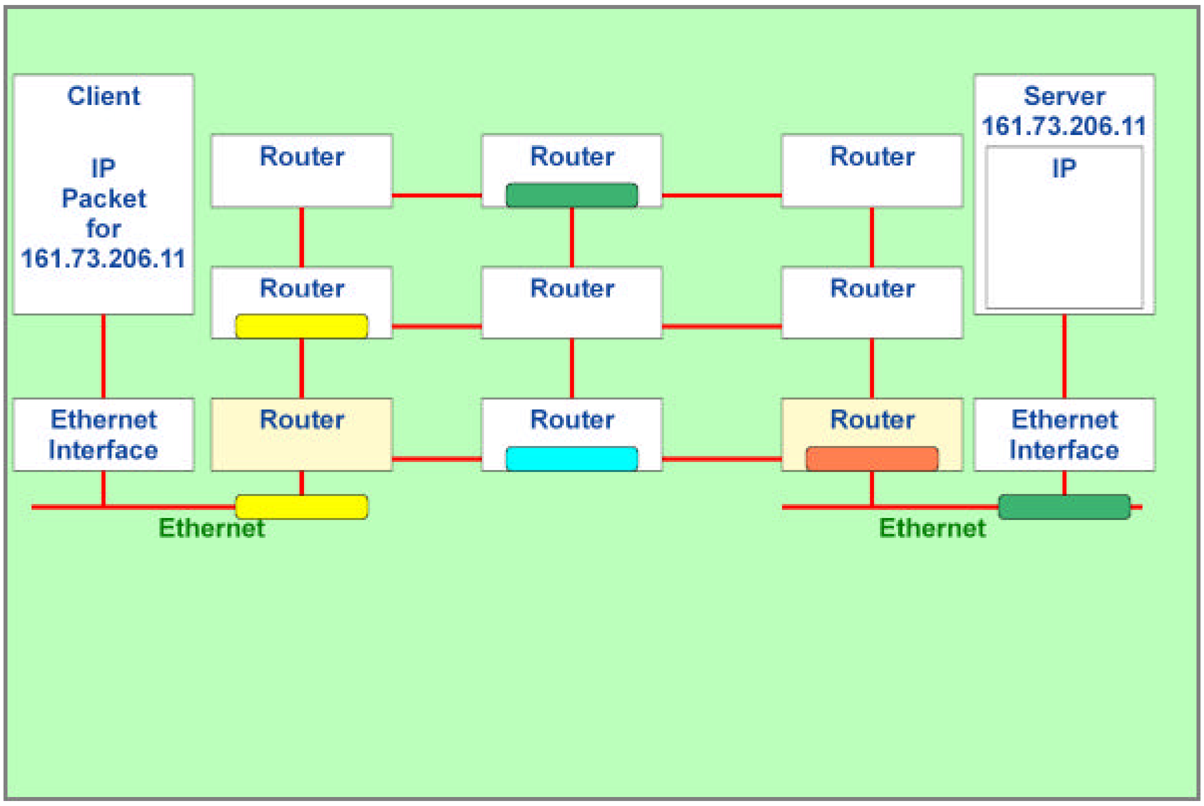


Figure 2.5: IP Packet Delivery

Figure 2.5 shows the progression of a set of packets through the Internet. The Client is almost certainly connected to some Local Area Network and it reaches the outside world via a router which has a number of options in trying to route the packet to W3C. Some of the packets will arrive at the remote site and the order will be unknown. Some will be lost en route. The next question is how do we achieve reliable transmission across the Internet. That is where TCP comes in.

3. Transmission Control Protocol

3.1 Introduction

The position of the Transmission Control Protocol (TCP) in the nature of things is shown in Figure 3.1. HTTP expects requests to arrive and responses also. TCP's role is to achieve that despite its underlying infrastructure (IP) being unreliable and unordered.

IP can be thought of as a set of buckets transmitting information from one place to another. The aim is for TCP to turn the individual buckets into a drainpipe [4].

TCP achieves this by establishing a unique path between the Client and Server (called a **socket**) and then ensuring that all packets are delivered, forcing a retransmission when packets appear to be lost. TCP does all the work in solving the problems of packet loss, corruption and reordering that the IP layer may have introduced.

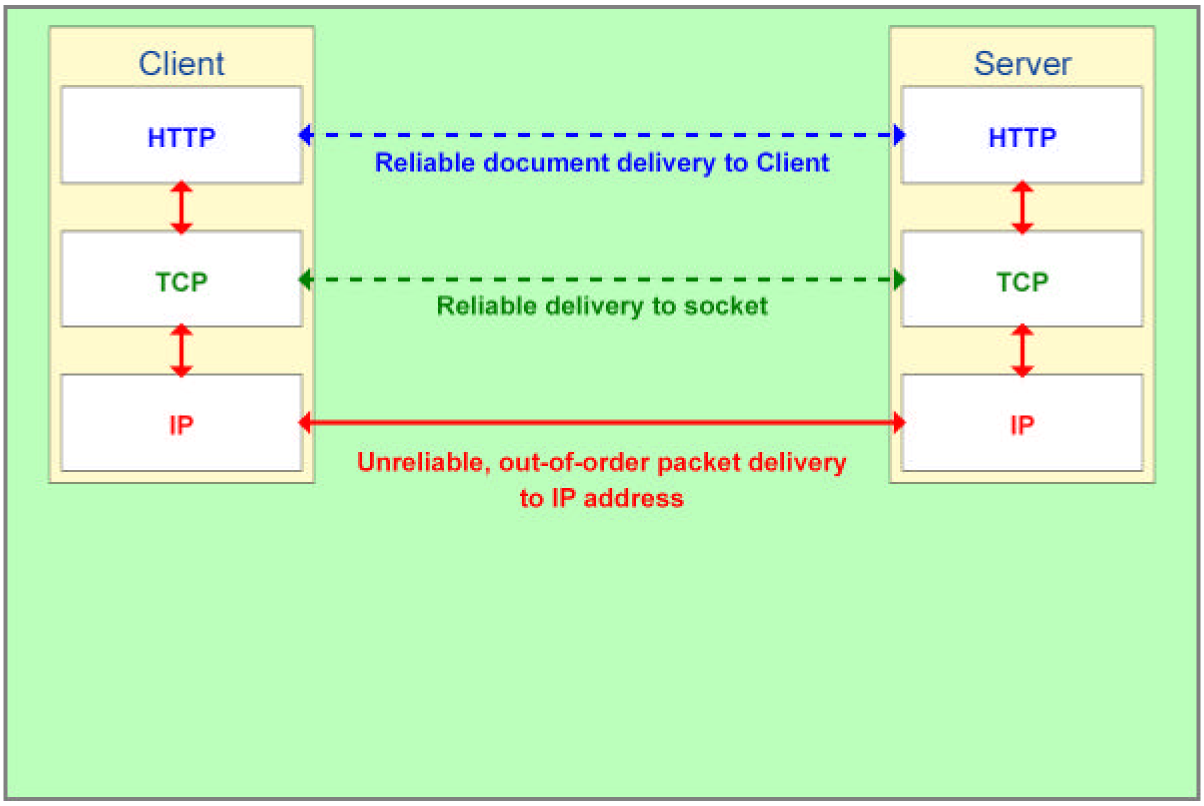

Client
HTTP
TCP
IP
Server
HTTP
TCP
IP
Reliable document delivery to Client
Reliable delivery to socket
Unreliable, out-of-order packet delivery to IP address

Figure 3.1: TCP's Role

3.2 Socket Creation

As we have seen, TCP can be thought of as a pipe delivering information reliably from one host to another. TCP and other transport level protocols use the concept of a **port**. As the Internet only provides one address for the computer, to carry on several separate transmissions at the same time, you need to differentiate between them and this is the role of the port. All transmissions of a certain type use the same port number as an extension of the internet addressing. Ports can be thought of as doors or letterboxes. A single house may have several doors or letterboxes if the house has been divided up for multiple occupancy.

To allow the Client and Server to believe that they are reading and writing files, the concept of a **socket** is introduced. Setting up a socket associated with a port allows the programs at each end to work as though they are reading and writing local files.

Figures 3.2 shows packets being delivered by IP. The name of the protocol using the packet is contained within the packet so it is easy to route the packets destined for TCP one way while those destined for UDP go another route. The packets arriving at the TCP application may be destined for any number of different transactions. Quite a few Clients will be requesting documents from the W3C web site.

Initially a unique connection needs to be set up between the Client and the Server. Many different uses of TCP may be taking place and there is a need to establish a connection for the transmission of HTTP. By convention port 80 is used for transmitting HTTP. At the Server end there is effectively a **bell** associated with port 80 that can be rung by the remote Client to say that it wants to establish a port 80 connection. The browser at the Client end may allow the user to download files and send mail as well. In this case there may be multiple ports opened by the single Client using different ports on the Server.

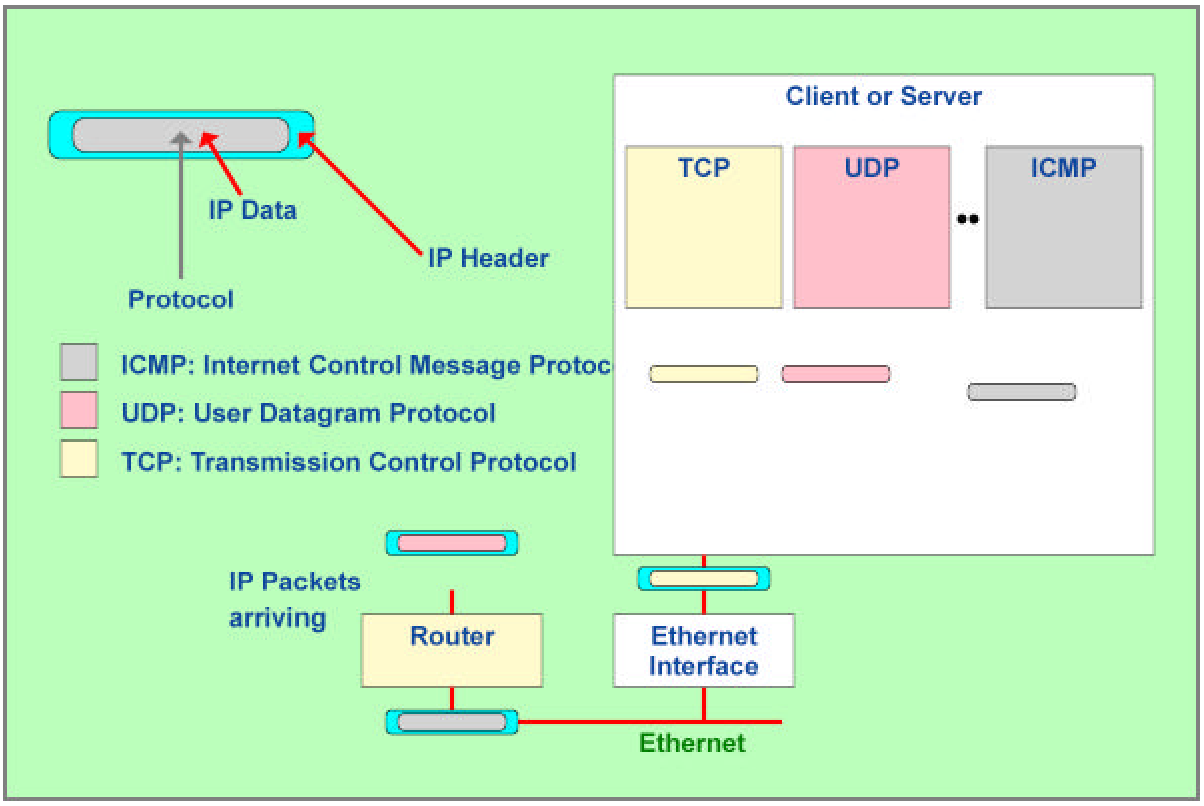


Figure 3.2: Selecting an Application

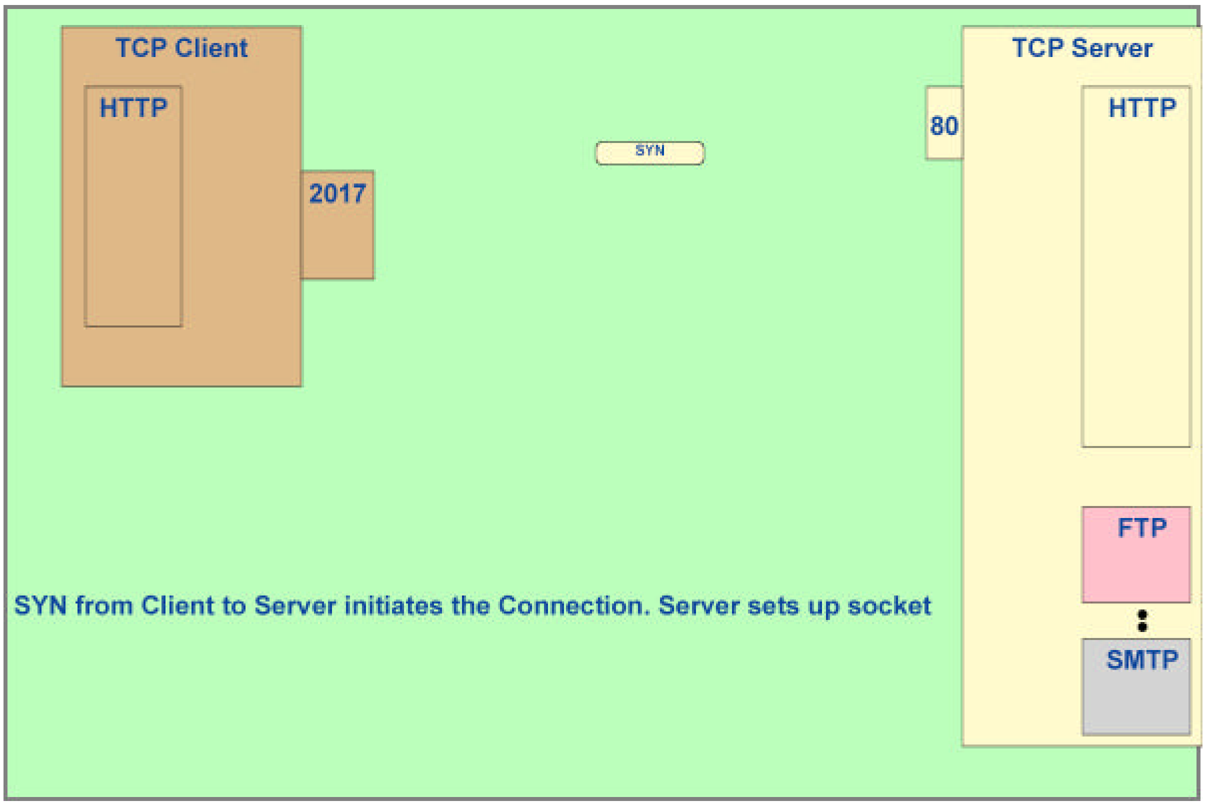

TCP Client
HTTP
2017
SYN
80
TCP Server
HTTP
FTP
SMTP
SYN from Client to Server initiates the Connection. Server sets up socket

Figure 3.3: Establishing a TCP Connection

Figure 3.3 shows the Client sending an IP packet enclosing a TCP packet that **rings** the bell to request a connection. The Client sets up a Port with a unique number (2017 in this case) before sending the TCP SYN packet. on receiving it, the Server sets up a real version of the port at its end (usually a real port 80, not just a door bell but a real door!) to allow transmissions to go from the Client to the Server. Figure 3.4 shows the port established and a SYN/ACK packet being send back to the Client. this acknowledges that the left-to-right transmission has been set up and also requests a connection in the reverse direction. Once this arrives at the server, the path is established and the Client sends back an ACK to acknowledge the reverse transmission path (Figure 3.5). Figure 3.6 shows the final position. Two connections have been established to send information from the Client to the Server and from the server to the Client. Three packets have been sent to achieve this.

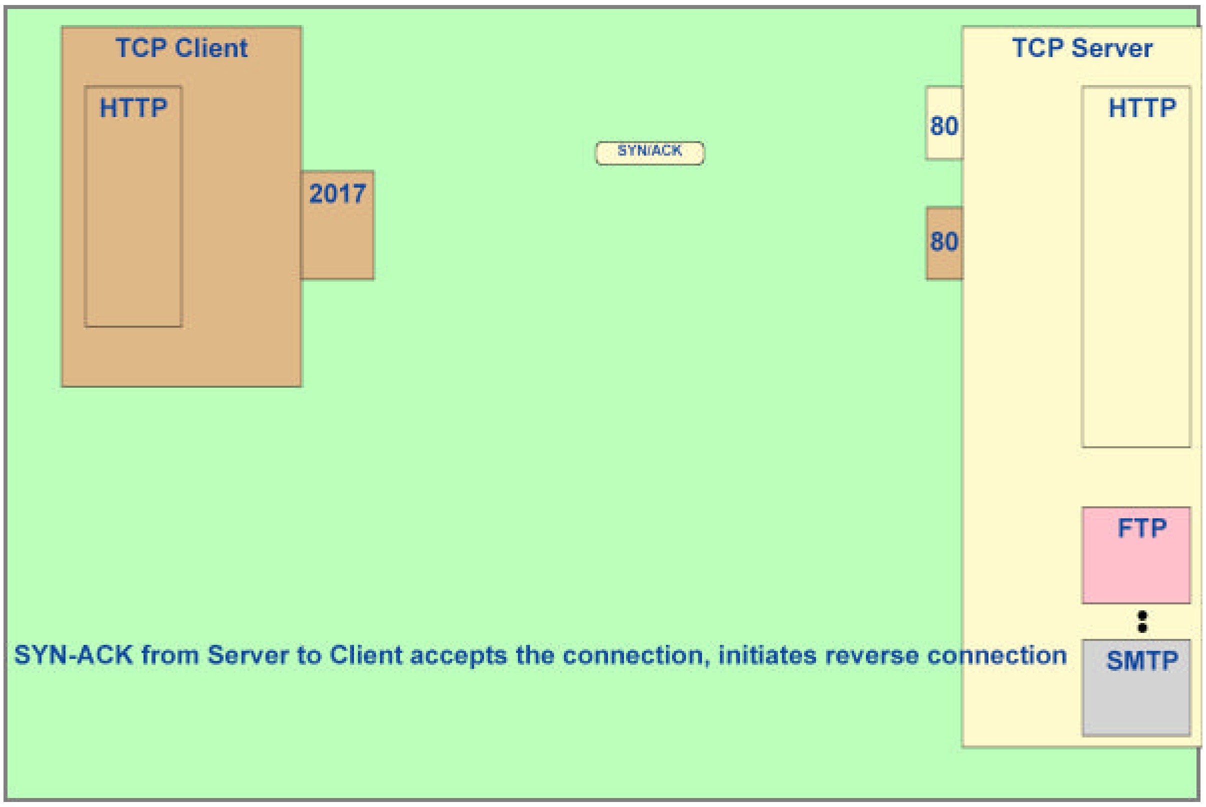

TCP Client
HTTP
2017
SYN/ACK
80
80
TCP Server
HTTP
FTP
SMTP
SYN-ACK from Server to Client accepts the connection, initiates reverse connection

Figure 3.4: Establishing a TCP Connection

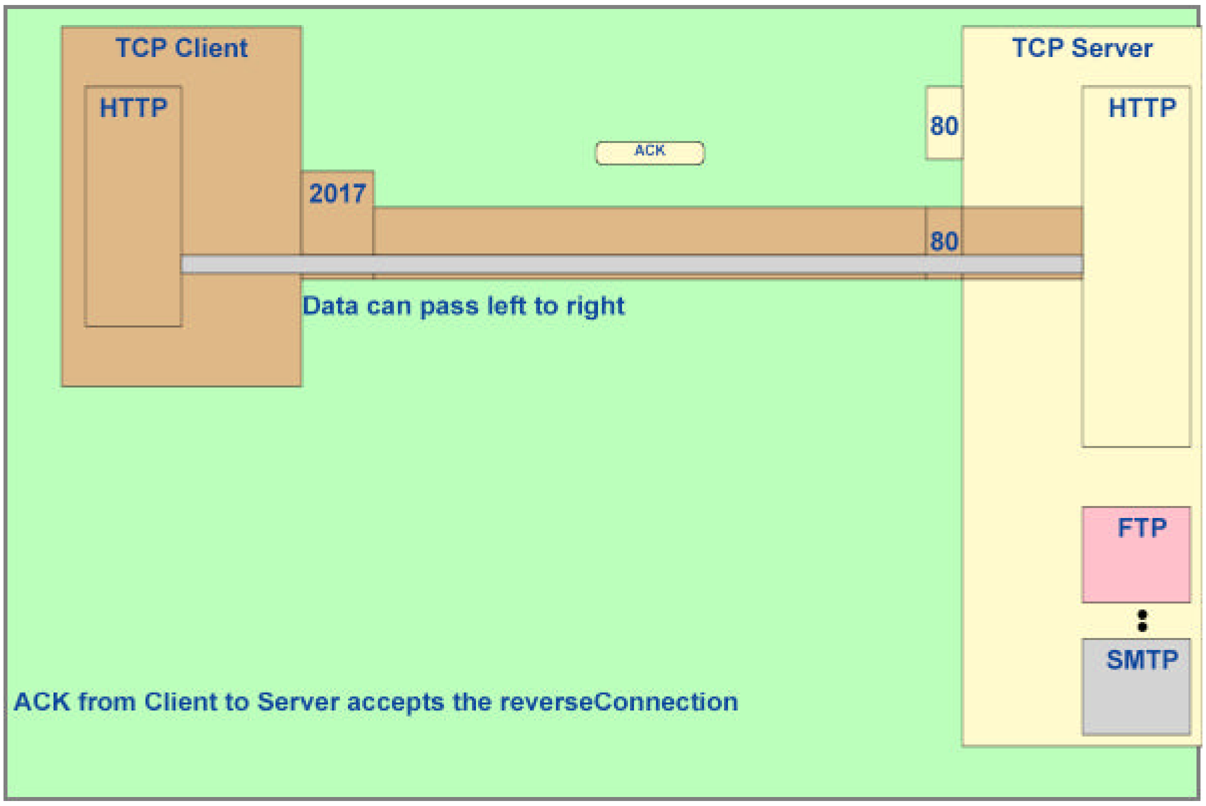

TCP Client
HTTP
2017
ACK
80
TCP Server
HTTP
80
Data can pass left to right
FTP
SMTP
ACK from Client to Server accepts the reverseConnection

Figure 3.5: Establishing a Connection

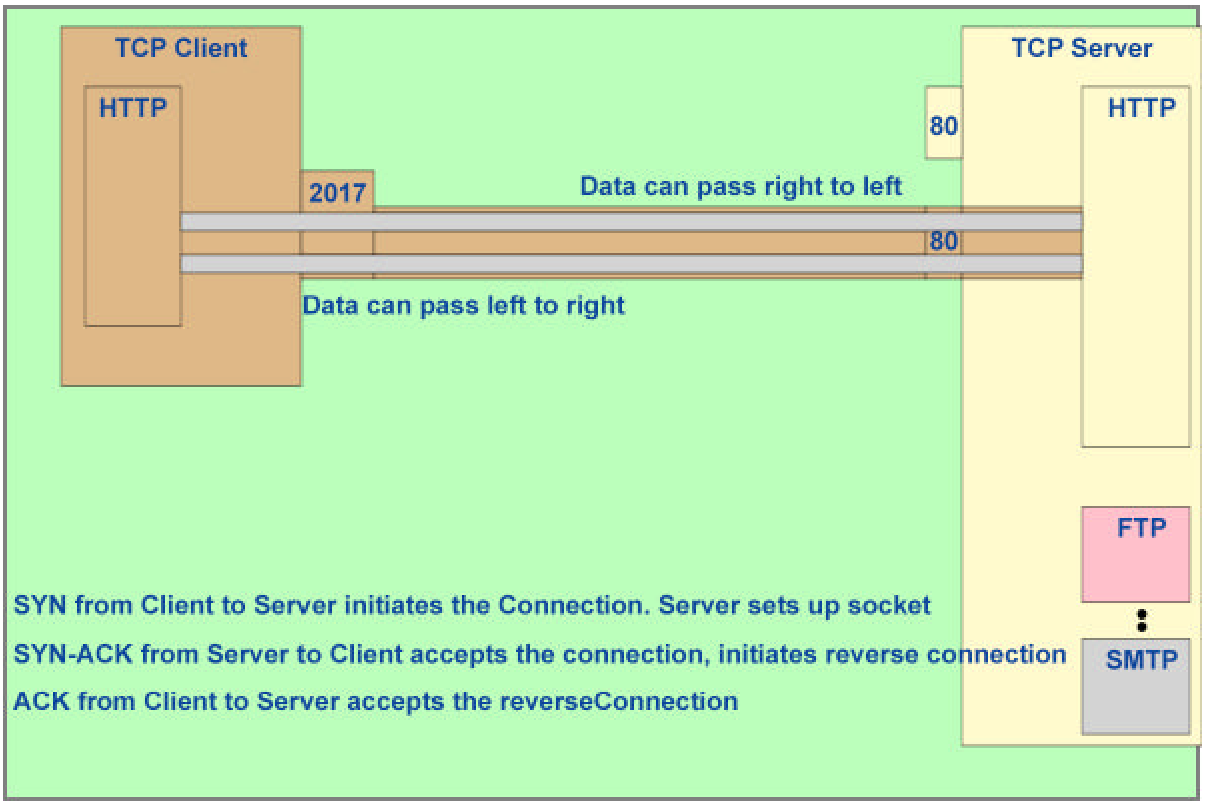
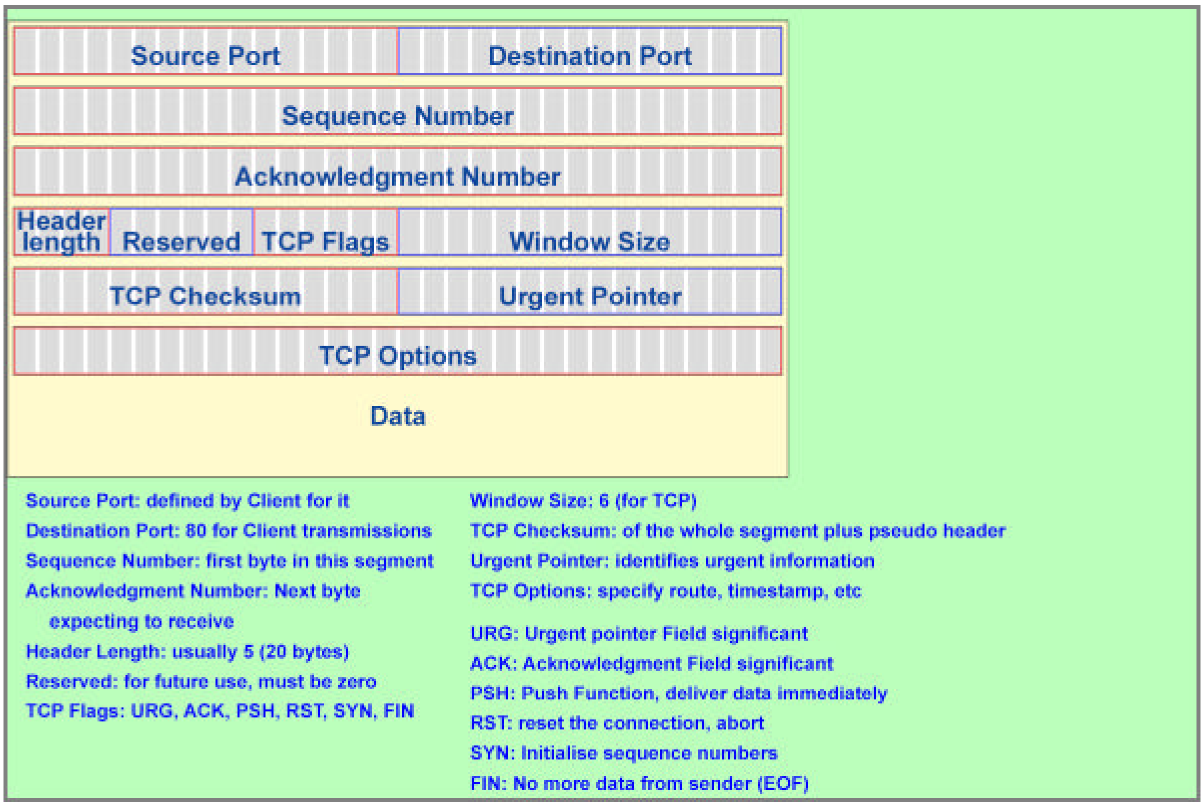

TCP Client
HTTP
2017
80
TCP Server
HTTP
Data can pass right to left
80
Data can pass left to right
FTP
SMTP
SYN from Client to Server initiates the Connection. Server sets up socket
SYN-ACK from Server to Client accepts the connection, initiates reverse connection
ACK from Client to Server accepts the reverseConnection

Figure 3.6: Establishing a TCP Connection

Once the connection is established, TCP can get down to the problem of delivering HTTP requests from Client to server and sending back the requested document.

3.3 Reliable Packet Transmission

Figure 3.7 shows the header for the TCP packet. The set of TCP Flags are used to indicate special TCP transmissions like the SYN, SYN/ACK and ACK transmissions needed to establish the connection. Note that the source and destination addresses do not appear as they can be ascertained from the IP header. The important field to ensure all transmissions take place is the Sequence Number which says which transmission this is in a sequence and the Acknowledgment Number which says what is the last packet in the sequence that has been correctly transmitted and received.



Source Port: defined by Client for it
Destination Port: 80 for Client transmissions
Sequence Number: first byte in this segment
Acknowledgment Number: Next byte expecting to receive
Header Length: usually 5 (20 bytes)
Reserved: for future use, must be zero
TCP Flags: URG, ACK, PSH, RST, SYN, FIN

Window Size: 6 (for TCP)
TCP Checksum: of the whole segment plus pseudo header
Urgent Pointer: identifies urgent information
TCP Options: specify route, timestamp, etc

URG: Urgent pointer Field significant
ACK: Acknowledgment Field significant
PSH: Push Function, deliver data immediately
RST: reset the connection, abort
SYN: Initialise sequence numbers
FIN: No more data from sender (EOF)

Figure 3.7: TCP Packet

Figure 3.8 shows a TCP transmission from Client to Server requesting a document to be transmitted. In the diagram, the ACK package waits to be transmitted as soon as the request arrives. Once the request is understood, the relevant HTML page is downloaded (see Figure 3.9) and split into IP packets ready for transmission with the appropriate TCP header as part of the data inside the IP packet.

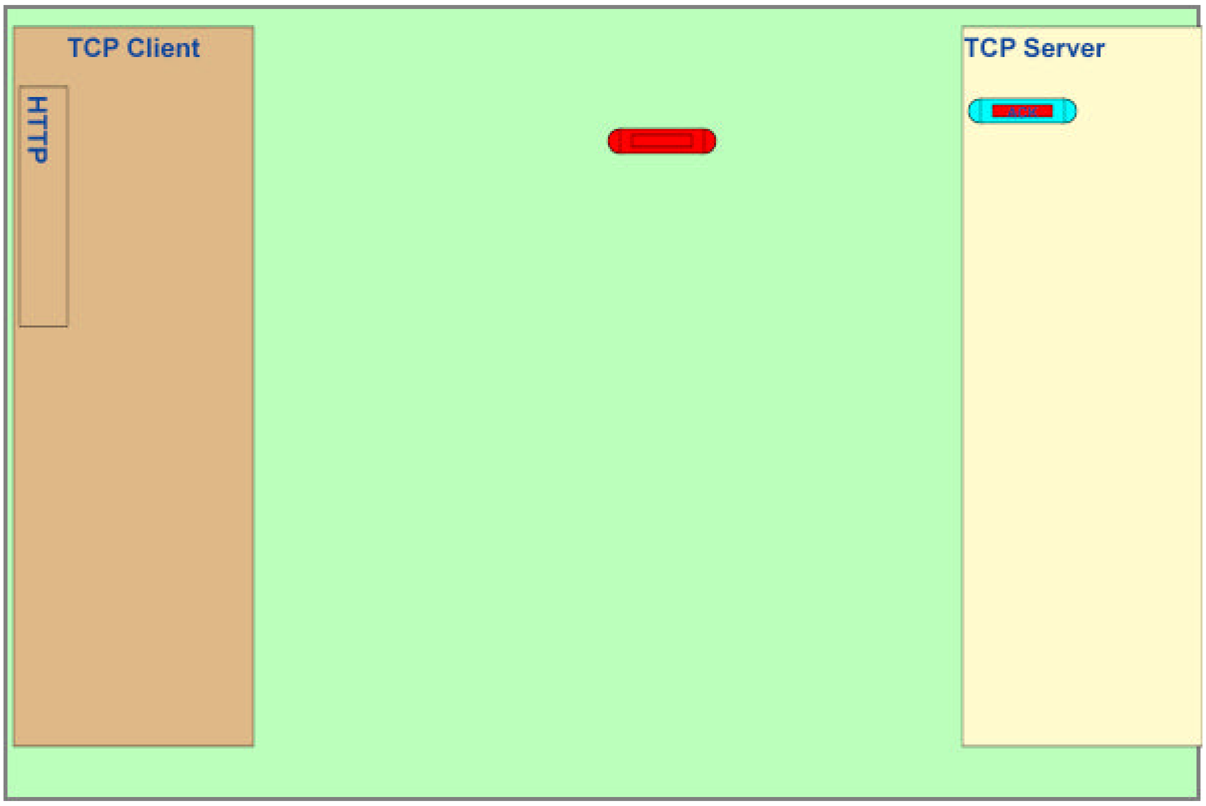


Figure 3.8: HTTP Request

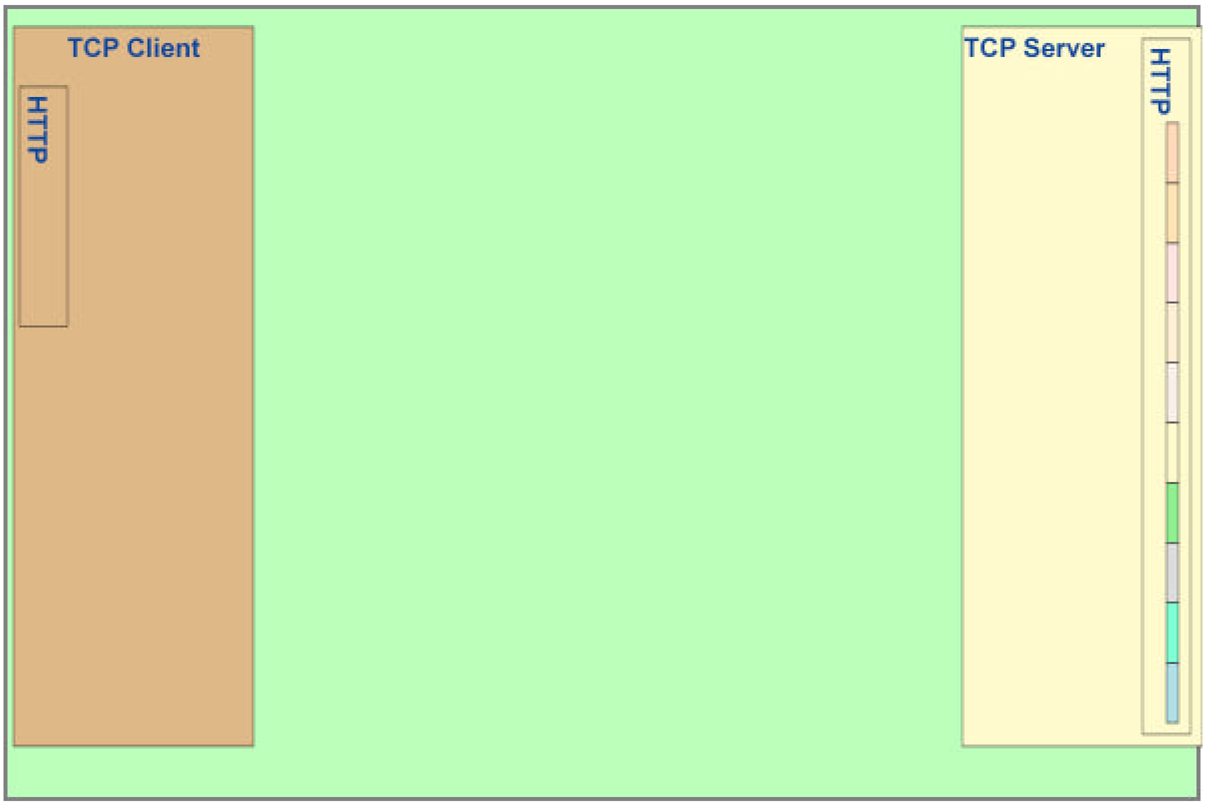


Figure 3.9: HTML Document to Return

Figure 3.10 shows the start of the transmission of the HTML document which fits into 10 TCP segments that are transmitted in 10 IP packets. The first packet has been sent and further packets will be sent although not all of them immediately. TCP will attempt to control the rate at which packets are sent to a rate it thinks the network can stand. So a few packets will be sent. Let us assume that the first packet sent is the first to arrive in which case the ACK returned indicates that of the sequence of 10 packets, the first has been successfully received.

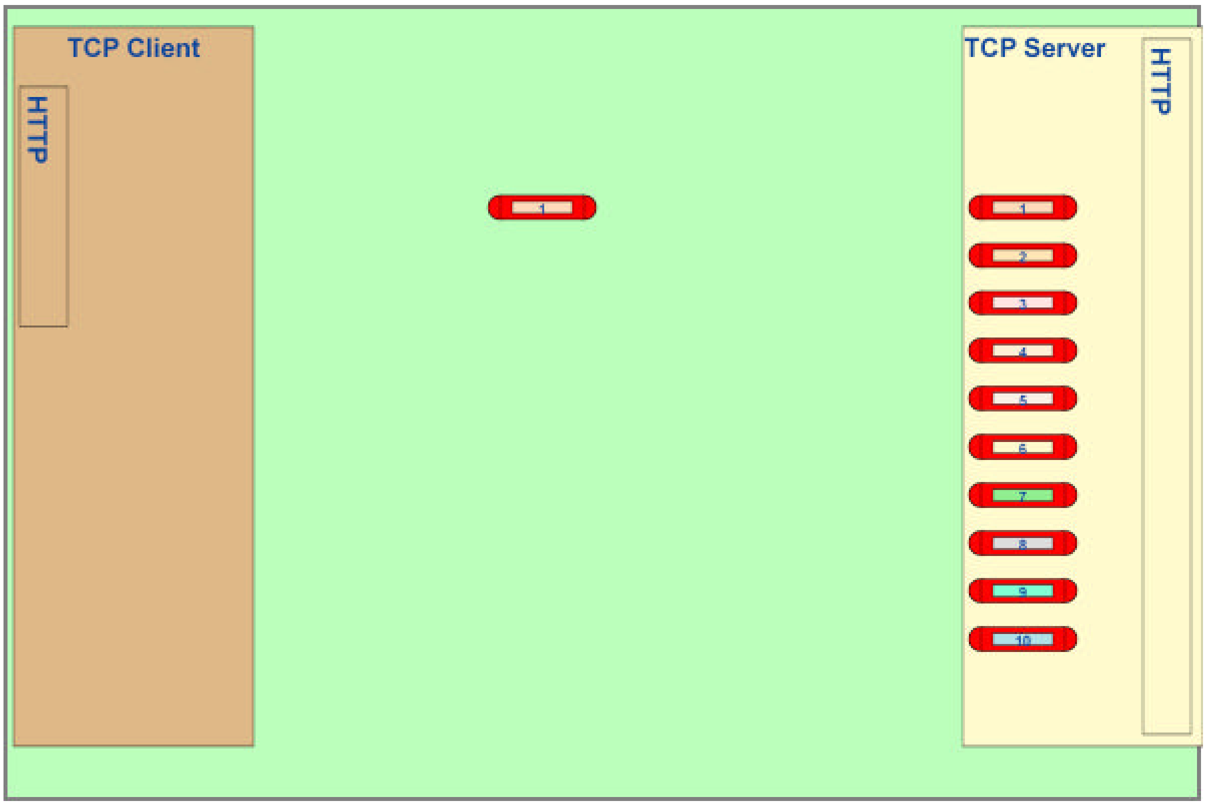


Figure 3.10: Starting to Send the Document

Figure 3.11 shows the position slightly further on. Packets 1 and 3 have arrived but 2 is still on its way. The ACK to package 3 states that the number of packets received in sequence is still just packet 1. There is no need to worry at this stage. Packet 2 will probably arrive. There is certainly no reason to be hasty and retransmit packet number 2.

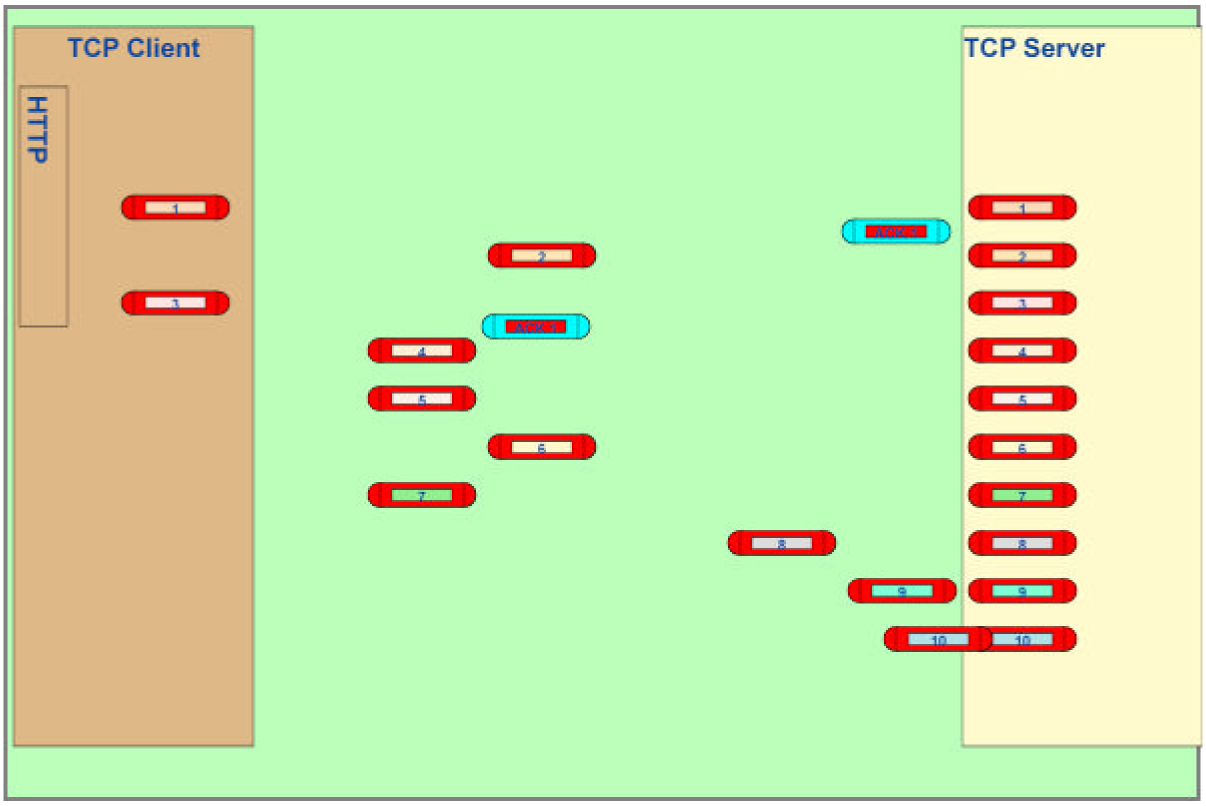


Figure 3.11: Some Packets Acknowledged

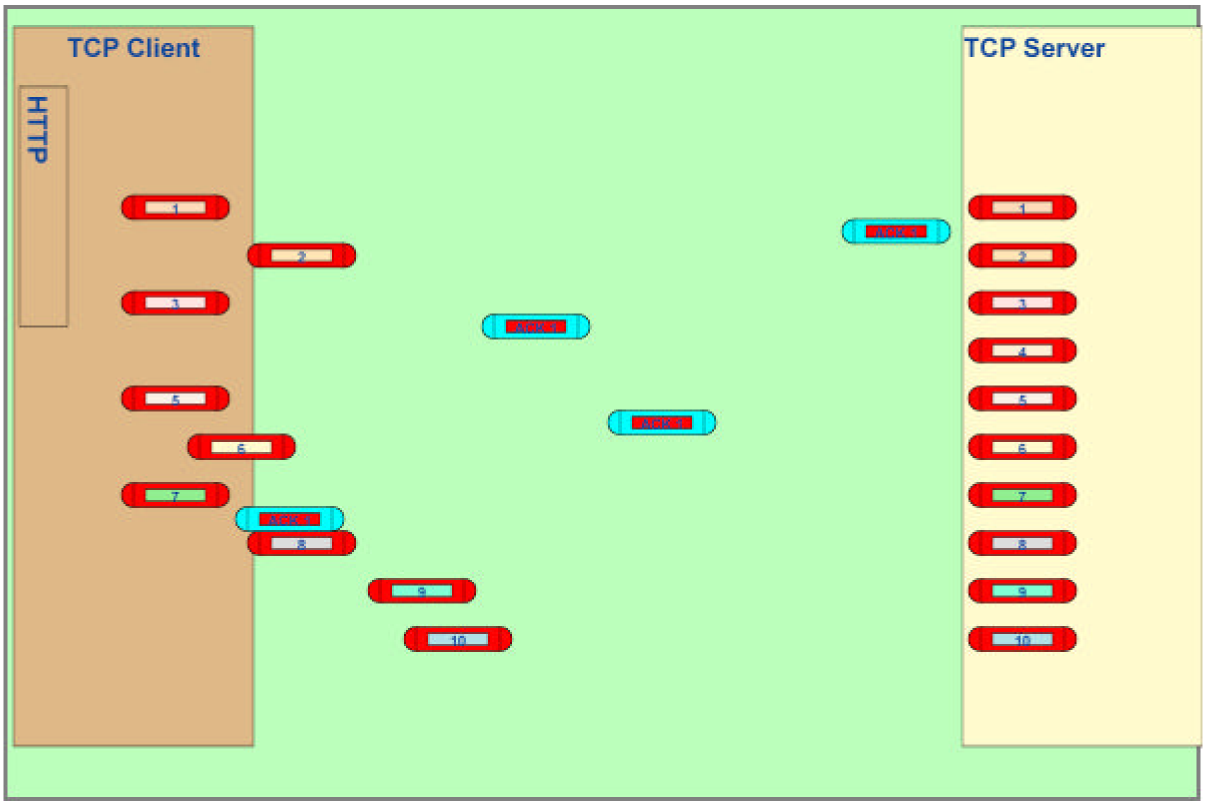


Figure 3.12: Discovering Lost Packets

Figure 3.12 shows the situation much further along. Packets 1, 3 5 and 7 have arrived and 2 is about to arrive. All four ACKs have shown that the sequence defined so far that has arrived successfuly is just the first packet. However if packet 2 arrives next, all of 1, 2 and 3 will have arrived so that the ACK for packet 2 will indicate that packets 1 through 3 have been successfully delivered. Packet 4 has been lost completely and no longer appears. At some stage as the following packets arrive and the situation is still set with only the first three arrived, the transmitter will realise that packet 4 is lost and will retransmit it.

The numbering of the packets and the acknowledgments indicating the number successfully completed in sequence is sufficient to ensure that all packets are delivered successfully. When the last packet arrives, the ACK will indicate that all the 10 transmissions have arrived successfully and the Server is then able to close the transmission. The closure actually takes one more transmission than the connection. With packets being delayed and retransmitted, it is possible for the same packet to arrive twice and for packets to arrive after the Client has indicated that all packets have been received so stopping is not completely straightforward.

4. HTTP/0.9 and HTTP/1.0

4.1 Introduction

Given the underlying TCP/IP network, it can be assumed that requests made by HTTP will arrive at the correct Server and that if the server locates the page required, it will be returned correctly. If the page is not located, an error will be received. It is on this basis that we will now look at the HTTP protocol.

The time line for HTTP is as follows:

December 1990
HTTP defined to transmit first Web pages
January 1992: HTTP/0.9
Simple GET protocol for the Web, limits on data transfer
March 1993: HTTP/1.0 Draft
Several iterations
May 1996:HTTP/1.0 RFC 1945
Headers give information about the data transfered. Greater data transfer in both directions
January 1997: HTTP/1.1 Proposal
Supports hierarchical proxy servers, caching, and persistent connections
June 1999: HTTP/1.1 Draft Standard
Initially proposed in 1997, significant use by 1998
2001: HTTP/1.1 Standard
Date to be defined

IETF, the organisatiion that standardises HTTP, tends to issue a prelease version of its standards with a **/0.9** label before the fulll standard is given the label **/1.0**. As can be seen progress has been relatively slow. It took several years before HTTP/0.9 was finalised and another 4 years before HTTP/1.0 was finalised. The latest version of HTTP is HTTP/1.1 and this is in wide use yet it is not formally a standard yet. This is generally the case with IETF. Much user experience is built up before the standard is finalised. By 1998, the world was nearly all using HTTP/1.0 with only 10% still using HTTP/0.9. Similarly, HTTP/1.1 is the main version in use today even though the standard is not finalised yet.

HTTP is based on **messages** that pass from the Client to the Server called **request** messages and those from the Server to the Client which are called **response** messages. All request messages start with a **request** line and all response messages start with a **response** line. Following the initial line in both cases it is possible to include zero or more **header** lines that give additional information.

HTTP/0.9 had a single GET request defined and no headers were defined. HTTP/1.0 added HEAD and POST requests and introduced headers.

4.2 GET Request

HTTP/0.9 had a very simple GET request message:

```
GET url CRLF
```

The HTTP request message consists of an ASCII string consisting of the message name followed by a URL and the Carriage Return and Line Feed characters. This can be simply demonstrated by opening a Telenet session on Port 80 of a web site as follows:

```
telnet www.w3.org 80
GET /www/overview.htm CRLF
```

This will cause the W3C site to send the document www.w3.org/www/overview.html to the command line interface. Something like the following will be returned:

```
<HTML>
. . .
</HTML>
```

Trying that example today may require:

```
telnet www.w3.org 80
GET /www/overview.htm HTTP/0.9 CRLF
```

Early on it was not necessary to say which version of HTTP was being used. With several possibilities now existing, it is best to indicate the type of HTTP request being made.

When the user clicks on a link in an HTML page, asks the browser to load a page, hit the back button or choose a favourite site, in all these cases the result is a GET messaage requesting the required page being sent by TCP to the Server having firsty established a connection. IN HTTP/0.9, the document returned was limited to 1024 characters. In consequence, most HTTP requests consisted of a single message/packet in each direction once the connection had been established. In HTTP/0.9 and HTTP/1.0, once the server has sent the document, the connection is dropped. No state is retained in HTTP concerning the transaction. In HTTP/0.9 there would have been 3 packets used to set up the connection, two packets used to send the request message (plus the acknowledgment), two to send the response and 4 to drop the connection, a total of 11 packets for the two information packets to be sent and received. This is of course assuming that no packets were lost or retransmitted.

4.3 Headers

The general message request format was extended in HTTP/1.0 to include headers:

Request line
Headers (0 or more lines)
Blank line
Optional Message Body

For example, the GET request might be:

```
GET /Protocols/myimage.png HTTP/1.0
If-Modified-Since: Monday, 13-Sep-99 08:00:00
```

The GET request line is followed by a header that modifies the request to say only retrieve the file if it has been modified since September 1999.

The format of the reply will be:

HTTP-version response-code response-text
headers (0 or more lines)
<blank line>
<returned file>

An example might be:

```
HTTP/1.0 200 Document follows
Date: Mon, 13 Sep 1999 08:00:00 GMT
Server: xxxx
Content-type: image/png
Last-modified: Sun, 12 Sep 1999 16.15.23 GMT
Content-length: 6000

<The png image>
```

The response has come via HTTP/1.0. The response code is 200 which says that the document was successfully located and will be following. The headers give additional information about the Server and the document.

Headers fall into four types:

1. **General** (Date, Pragma)
2. **Request** (Authorization, From, If-Modifed-Since, User-Agent)
3. **Response** (Location, Server, WWW-Authenticate)
4. **Entity** (Allow, Content-Encoding, Content-Length, Content-Type, Expires, Last-Modified

This is not the complete list available with HTTP/1.0 but gives a flavour of those available. They all have the general format of a name followed by colon and then the information. Request headers only appear in request messages and response headers in replies. General and Entity can appear in both. Brief descriptions are:

Header	Type	Meaning
Date	General	Current time/date
Pragma	General	Request to behave in a certain way (**no-cache** requests proxies not to provided cached copies)
Authorization	Request	Send userid/password
From	Request	User sends email address as identification
If-Modified-Since	Request	Conditional GET. Ignore request if not modified since
User-Agent	Request	Web browser name and version number
Location	Response	Redirect the request to where it can be found
Server	Response	Type of server
WWW-Authenticate	Response	Challenge client seeking access to resource that needs authentication
Allow	Entity	Defines methods allowed to access resource
Content-Length	Entity	Number of bytes of data in body
Content-Type	Entity	MIME-type of data
Content-Encoding	Entity	Decoding needed to generate Content-Type, usually used for compression
Expires	Entity	When to discard from cache
Last-Modified	Entity	Time when data last modified

4.4 Responses

The responses all start with the response line which includes the response code. These are divided into the following classes:

- 1XX: Information
- 2XX: Request successful
- 3XX: Client error
- 4XX: Server error
- 5XX: System failure

No information responses were defined in HTTP/1.0 although the class was set up. The possible responses are:

Response	Meaning
200	Request succeeded
202	Request accepted, processing incomplete
204	No Content, for example clicking on part of an image map that is inactive
301	Requested URL assigned a new permanent URL
302	Requested URL temporarily assigned a new URL
304	Document not modified. Not changed since the last modification time in a request
400	Bad request
401	Request not accepted, need user authentication
403	Forbidden. Reason might be given in an entity response
404	Not found, the **most widely received message**
500	Internal server error
501	Not implemented
502	Invalid response from gateway or upstream server.
503	Service temporarily unavailable

The most frequent ones are 200 and 404.

4.5 HEAD and POST Requests

Two new requests were added to HTTP/1.0, HEAD and POST. HEAD is similar in format to GET except that it does not expect the file to be transmitted. All it requires are the associated header lines. For example:

```
HEAD /Protocols/Papers.html HTTP/1.0
If-Modified-Since: Monday, 13-Sep-99 08:00:00
```

This is a request to see if the document has been modified since a specified time on 13 September 1999. The response would be something like:

```
HTTP/1.0 304 Not modified
```

The POST command gave the opportunity for a more substantial document to be sent to the Server with the request. The most frequent use of POST is with form filling in HTML (and will be discussed in detail there). In this case several lines of input may need to be sent to the Server for processing. The format of the POST command is:

```
POST  URI  HTTP-version
headers (0 or more lines)
<blank line>
body
```

A specific example might be:

```
POST /Protocols/Papers.html HTTP/1.0
Content-length:  28

name=Bob+Hopgood&children=3
```

5. HTTP/1.1

5.1 Introduction

HTTP/1.1 is a major update to HTTP/1.0. A sub-set of the new Headers in HTTP/1.1 are:

Header	Type	Meaning
Cache-Control	General	Caching information
Connection	General	Connection management
Trailer	General	Headers at the end of the message, used with chunking
Transfer-Encoding	General	Transformation applied to message body, allows separate chunks to be sent
Upgrade	General	Suggesting another newer protocol server can handle
Via	General	Information about intermediate servers passed on
Warning	General	Non-catastrophic error
Proxy-Authorization	Request	Authentication with proxy
Accept	Response	Preferred media type
If-Match, If-None-Match	Request	Checking Entity Tags
If-Unmodified-Since	Request	Check when last modified
Expect	Request	Expected server behaviour, can it handle a large file
Host	Request	Resource Host, mandatory in HTTP/1.1
Max-Forwards	Request	Limits number of hops
Range	Request	Requests part of file
Location	Response	Alternative place to find file
Retry-After	Response	Time before retrying
Accept-Ranges	Response	Server can accept range requests
Proxy-Authenticate	Response	Authentication but for the proxy
ETag	Response	Defines ETag
Vary	Response	Variant of Resource
Content-Language	Entity	Language of resource
Content-Location	Entity	Alternative location
Content-Range	Entity	Range in resource

The design of HTTP/1.0 had several limitations once the Web started to be used in a wider context. The simple model of setting up a connection, downloading a page, and then dropping the connection does not make a great deal of sense once images start to be included in a document. Several GET requests are needed to retrieve all the parts of the page. Many of these will be from the same Server resulting in several connections being set up and destroyed for a single page to be transmitted. TCP keeps some information concerning bandwidth congestion around the network so that future transmissions can benefit from past experience. Such information is lost when the connection is broken and has to be discovered again when the connection is reestablished. Browsers started to set up multiple connections at the same time to try and improve performance for one user while making things worse for others.

A second major problem occurs once caches became common place. There is little information available as to when an old stale page can be tolerated and when an up-to-date page must be provided. It should be possible to receive a Web page from a cache rather than the original server and be confident that it is the most recent.

There was also the need for many minor enhancements within HTTP/1.1 and we will discuss one or two of these once we have looked at persistent connections and caching. Five new requests, over 30 new headers, and another 20 response codes have been added to HTTP/1.1 just to give a feel for the level of change.

5.2 Persistent Connection

A fundamental difference between HTTP/1.0 and HTTP/1.1 is that the connection is not dropped when the request has been replied to. In HTTP/1.1, the connection remains open until the Server makes a decision that it is no longer required. This is a heuristic decision based on the availability of resources and the use that has been made of the connection. HTTP/1.1 also allows several GET requests to be sent over the connection before the first response has been returned. At the server end, this allows the server to return some documents before others depending on whether it needs to access them from disc or whether they are available in its own cache.

As well as improving users perceived performance, because the connections are not dropped and TCP can make more efficient use of the network, it reduces the load on both the network and the servers. An estimated improvement is that downloading a similar file with HTTP/1.1 requires about 60% of the resources in the Server compared with HTTP/1.0. In hardware terms for a large site, that might mean that 20% less servers are needed.

W3C has a benchmark that originated by merging the Netscape and Microsoft home pages as they were a few years ago complete with a set of images and using this Microscape benchmark to check out performance. There were 43 images in the combined **Microscape** page. Four secenarios were compared:

- HTTP/1.0: using 4 simultaneous connections
- HTTP/1.1: using 1 persistent connection
- HTTP/1.1 pipeline: using 1 persistent connection
- HTTP/1.1 pipeline + compression: using 1 connection

The first uses HTTP/1.0 and opens 4 TCP connections to bring down the images with the page itself. The second uses HTTP/1.1 but with a single connection that is persistent. The third pipelines the requests rather than wait for each to complete. the last test also added compression which is available in HTTP/1.1. This will not improve the images very much as they are already compressed but has some effect on the page itself. the results are shown in Figure 5.1.

As can be seen, the four HTTP/1.0 connections are faster than the single HTTP/1.1 connection but at the cost of many more TCP packets being sent due to the dropping of the connections after each transfer. Once the HTTP/1.1 connection is pipelined, the number of packets needed goes down and the performance is significantly better. With the test with many images, only marginal improvement comes from the compression.

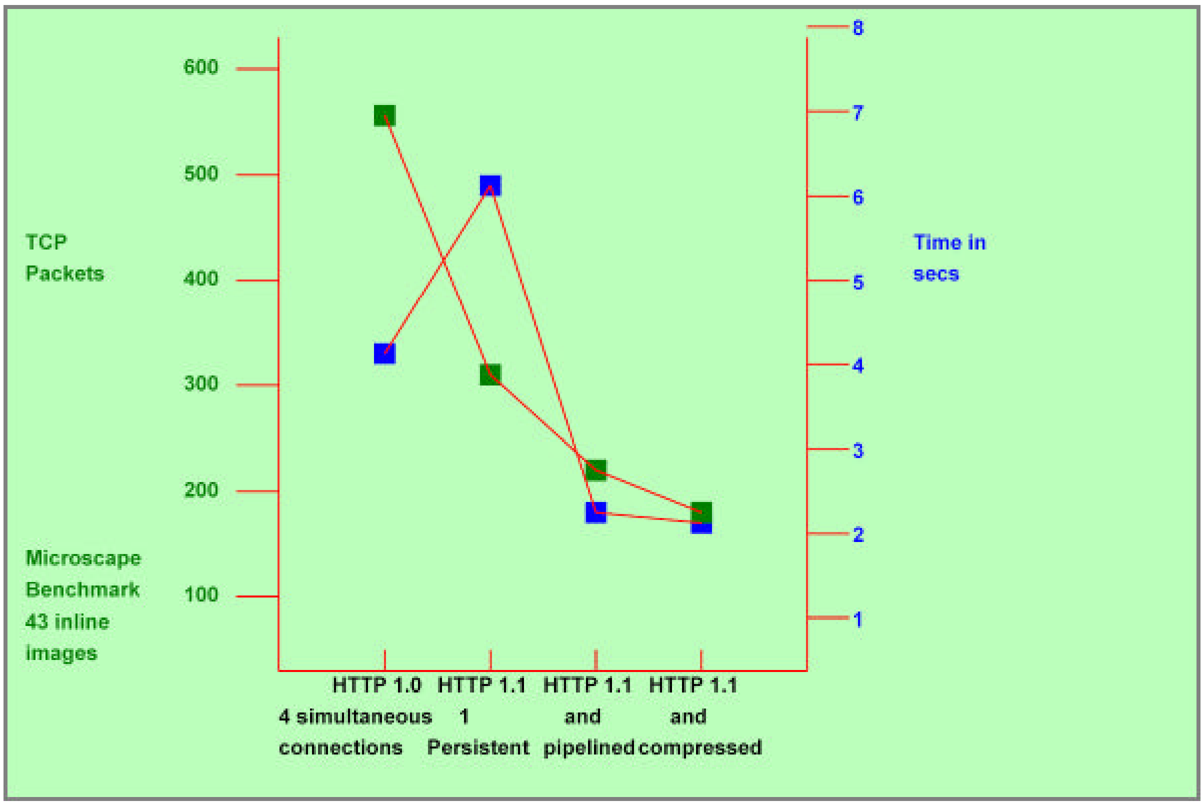


Figure 5.1: Microscape Benchmark

5.3 Caching

The **Expires** header in HTTP/1.0 does help in recognising when a cached page is no longer valid and has gone stale. Much more control is provided in HTTP/1.1 with the **Cache-Control** header. This allows either the Server to:

- Declare what is cacheable
- Say what may be stored by caches
- Provide a more powerful expiration mechanism
- Provide support for revalidation and reload

A simple use might be:

```
Cache-Control: no-cache
```

This would tell all the caches that this page should not be cached presumably because it contains information that quickly ggoes out-of-date. The full set of possibilities in GET requests are:

Directive	Description
no-cache	Must come from original server
only-if-cached	Only from cache
no-store	Do not cache response
max-age	Age no greater than this. max-age=0 same as no-cache
max-stale	Can be expired but not more than this. max-stale=120 accepts stale response as long as not by more than 2 minutes
min-fresh	Fresh for at least this long. min-fresh=120 means that response has to have an expired time at least 2 minutes in the future
no-transform	Proxy must not change media type. Not allowed to turn a png into a gif

Support is also provided for the server to give further information when it returns that page using the Cache-Control header:

Directive	Description
public	Insists page is cacheable. Cache might have thought it was in a non-cacheable class
private	Stops a response being cacheable (such as Customer ID or a password)
no-store	Stops caching of both request and response
no-cache	Insists page is not cacheable. Proxy may cache as long as it validates each time
no-transform	Must not transform MIME type
must-revalidate	Strictly follow the rules. Must revalidate. If it cannot, must return a 504 or 500 error
proxy-revalidate	Proxy must follow the rules. Local browser caches don't have to
max-age	Maximum time it is fresh, overrides Expires Header
s-maxage	Maximum time it should be cached in a proxy cache

5.4 Other Enhancements

There are many other enhancements in HTTP/1.1 but only two will be discussed in this Primer.

HTTP/1.0 allowed the freshness of a page to be validated with the If-Modified-Since request. This ensures that the page returned is fresher than a previous date but does not guarantee that it is the freshest nor does it give the user much idea whether the page it already has is up-to-date. Thus the user has little confidence that the page being viewed is the freshest. Also, to first order the time stamps are only to the nearest second so that if the information is rapidly changing, the mechanism does not work either.

HTTP/1.1 has introduced the Entity Tag or **ETag** for this purpose. An ETag uniquely identifies a copy of a page by giving it a unique identifier. Strong ETags change whenever the page changes. Weak ETags should change whenever the page changes. If the current version on the Server has the same ETag as the one in a cache then the two should be the same version of the page. A new request header has been added, If-Match that allows the user or a proxy to validate the copy of the page it already has for freshness.

A second problem that arises on the Web is when standard information pages are returned in a well-defined format but where the information changes regularly. For example, the latest prices on the Stock exchange are changing by the minute and if all are listed, even if it is just the FTSE100, the page is still quite long and most users will only wish to see a subset of the whole page. The Range request has been added to allow the user to only download part of a page between well defined **ranges**. For example:

```
GET /largefile.html HTTP/1.1
Range: 0-200, 5000-5030
```

This would return just the bytes specified. Some formats such as Adobe's PDF keep a n index at the front giving the range of each page in the document in terms of bytes from the start of the file. The user can use the first range request to pick up the index and then a subsequent range request to pick up the page or pages of interest. Servers are not required to handle range requests. The **Accept-Ranges** header has been added to allow the server to indicate that it is capable of handling range requests.

HTTP has also introduced the **Expect/Continue** request/response to help with the handling of large volumes of information. A user attempting to send a large POST file can send the length and ask the Server if it can continue. The Server can respond saying whether or not it can handle such a large file.

Appendix A

References

There are some useful Web sites and books relevant to HTTP:

1. http://www.w3.org/Protocols/rfc1945/rfc1945
   HTTP/1.0 May 1996
2. http://www.ietf.org/rfc/rfc2616.txt
   HTTP/1.1 Draft Standard, June 1999
3. Web Protocols and Practice, Balachander Krishnamurthy and Jennifer Rexford
   Addison Wesley, 2001.
4. The World Wide Web, Mark Handley and Jon Crowcroft
   UCL Press, 1996.
5. Computer Networks (Third Edition), Andrew S Tanenbaum
   Prentice-Hall, 1996.
6. High-Performance Communication Networks (2nd edition), Jean Walrand & Pravin Varaiya
   Morgan Kaufmann Publishers, 2000.

Appendix B: HyperText Transfer Protocol Design Issues, 1991 by Tim Berners-Lee

Here are some design decisions to be made for protocols for information retrieval for hypertext.

Underlying protocol

There are various distinct possible bases for the protocol - we can choose

- Something based on, and looking like, an Internet protocol. This has the advantage of being well understood, of existing implementations being all over the place. It also leaves open the possibility of a universal FTP/HTTP or NNTP/HTTP server. This is the case for the current HTTP.
- Something based on an RPC standard. This has the advantage of making it easy to generate the code, that the parsing of the messages is done automatically, and that the transfer of binary data is efficient. It has the disadvantage that one needs the RPC code to be available on all platforms. One would have to chose one (or more) styles of RPC. Another disadvantage may be that existing RPC systems are not efficient at transferring large quantities of text over a stream protocol unless (like DD-OC-RPC) one has a let-out and can access the socket directly.
- Something based on the OSI stack, as is Z39.50. This would have to be run over TCP in the internet world.

Current HTTP uses the first alternative, to make it simple to program, so that it will catch on: conversion to run over an OSI stack will be simple as the structure of the messages is well defined.

Idempotent ?

Another choice is whether to make the protocol idempotent or not. That is, does the server need to keep any state information about the client? (For example, the NFS protocol is idempotent, but the FTP and NNTP protocols are not.) In the case of FTP the state information consists of authorisation, which is not trivial to establish every time but could be, and current directory and transfer mode which are basically trivial. The proposed protocol IS idempotent.

This causes, in principle, a problem when trying to map a non-idempotent system (such as library search systems which stored "result sets" on behalf of the client) into the web. The problem is that to use them in an idempotent way requires the re-evaluation of the intermediate result sets at each query. This can be solved by the gateway intelligently caching result sets for a reasonable time.

Request: Information transferred from client

Parameters below, however represented on the network, are given in upper case, with parameter names in lower case. This set assumes a model of format negotiation in which in which the client says what he can take, and the server decides what to give him. One imagines that each function would return a status, as well as information specified below.

When running over a byte stream protocol, SGML would be an encoding possibility (as well as ASN/1 etc).

Here are some possible commands and parameters:

GET document name
Please transfer a named document back. Transfer the results back in a standard format or one which I have said I can accept. The reply includes the format. In practice, one may want to transfer the document over the same link (a la NNTP) or a different one (a la FTP). There are advantages in each technique. The use of the same link is standard, with moving to a different link by negotiation (see PORT).

SEARCH keywords
Please search the given index document for all items with the given word combination, and transfer the results back as marked up hypertext. This could elaborate to an SQL query. There are many advantages in making the search criterion just a subset of the document name space.

SINCE datetime
For a search, refer to documents only dated on or after this date. Used typically for building a journal, or for incremental update of indexes and maps of the web.

BEFORE datetime
For a search, refer to documents before this data only.

ACCEPT format penalty
I can accept the given formats . The penalty is a set of numbers giving an estimate of the data degradation and elapsed time penalty which would be suffered at the CLIENT end by data being received in this way. Gateways may add or modify these fields.

PORT
See the RFC959 PORT command. We could change the default so that if the port command is NOT specified, then data must be sent back down the same link. In an idempotent world, this information would be included in the GET command.

HEAD doc
Like GET, but get only header information. One would have to decide whether the header should be in SGML or in protocol format (e.g. RPC parameters or internet mail header format). The function of this would be to allow overviews and simple indexes to be built without having to retrieve the whole document. See the RFC977 HEAD command. The process of generation of the header of a document from the source (if that is how it is derived) is subject to the same possibilities (caching, etc) as a format conversion from the source.

USER id
The user name for logging purposes, preferably a mail address. Not for authentication unless no other authentication is given.

AUTHORITY authentication
A string to be passed across transparently. The protocol is open to the authentication system used.

HOST
The calling host name - useful when the calling host is not properly registered with a name server.

Client Software
For interest only, the application name and version number of the client software. These values should be preserved by gateways.

Response

Suppose the response is an SGML document, with the document type a function of the status.

Possible replies one could imagine, encoded as SGML:

```
<!GDOC HTML> a
normal HTML document <!/GDOC> <!GDOC HTML>
```

Document with preprocessing instructions:

```
<HEADER> normal HTML <FORMAT REP="Text/ASCII">
<FILTER PROCESS="crypt-md5" PROMPT="Delphi"> <FILTER PROCESS="uncompress">
<FILTER PROCESS="tar"> </HEADER>
asdfghjklsdfghjklasdfghjksdfhjkzxcvbnmeryuioxcvbj
ewft76t8yytcvncncnryxmry02zmxnxjxb7wdtx7b7rtwbt87
```

Status
: A status is required in machine-readable format. See the 3-figure status codes of FTP for example. Bad status codes should be accompanied by an explanatory document, possible containing links to further information. A possibility would be to make an error response a special SGML document type. Some special status codes are mentioned below.

Format
: The format selected by the server

Document
: The document in that format

Status codes

Success
: Accompanied by format and document.

Forward
: Accompanied by new address. The server indicates a new address to be used by the client for finding the document. the document may have moved, or the server may be a name server.

Need Authorisation
: The authorisation is not sufficient. Accompanied by the address prefix for which authorisation is required. The browser should obtain authorisation, and use it every time a request is made for a document name matching that prefix.

Refused
: Access has been refused. Sending (more) authorization won't help.

Bad document name
: The document name did not refer to a valid document.

Server failure
: Not the client's fault. Accompanied by a natural language explanation.

Not available now
: Temporary problem - trying at a later time might help. This does not i,ply anything about the document name and authorisation being valid. Accompanied by a natural language explanation.

Search fail
: Accompanied by a HTML hit-list without any hits, but possibly containing a natural explanation.

Valid