Chapter 5

Persistent-Connection HTTP

In this chapter, we present our initial solution to the performance problems that arise from the composition of HTTP and TCP. We call this solution *persistent-connection HTTP* (*P-HTTP*)[1]. P-HTTP introduces a new connection abstraction for HTTP, that of a *persistent connection* which is reused for multiple HTTP transactions. P-HTTP avoids the overhead introduced by multiple short-length connections in HTTP/1.0, and thereby reduces latency. It further reduces latency by *pipelining* multiple transactions over the persistent connection. The ideas of persistent connections and pipelining in P-HTTP, developed by us in 1994 [85], have been adopted by the new HTTP/1.1 protocol [30]. We focus on P-HTTP in this chapter, and touch upon only briefly HTTP/1.1.

We begin by enumerating the specific goals of P-HTTP in Section 5.1. We then discuss the design of P-HTTP, specifically persistent connections and pipelining, in Section 5.2 and Section 5.3. We describe our implementation in Section 5.4 and present performance results in Section 5.5. In Section 5.6, we discuss extensions of P-HTTP. In Section 5.7, we point out limitations of P-HTTP, which motivated us to develop the *TCP session* and *TCP fast start* techniques presented in Chapter 6 through Chapter 8. Finally, we present a summary in Section 5.8.

1. The name "P-HTTP" was picked much after this work was done by Jeff Mogul and me. The credit for the choice of this name goes to Jeff [74].

5.1 Goals of P-HTTP

In addressing the performance problems arising from the composition of HTTP and TCP, the specific goals are to:

1. Reduce latency arising from:

   a. Connection establishment. The latency is dominated by the RTT for the SYN handshake, but it also includes the overhead of connection setup processing at the end hosts.

   b. HTTP request-response exchanges.

   c. Slow start.

2. Reduce the dependence on timeouts for loss recovery.

3. Make congestion avoidance and control as effective as with a single TCP connection.

We discuss the two key components of P-HTTP that help in meeting these goals, persistent connections and pipelining, in turn.

5.2 Persistent Connections

Since many of the performance problems of HTTP/1.0 arise from the use of a large number of short-length TCP connections, we eliminate the underlying cause by replacing the numerous short connections with a single *persistent* TCP connection. Multiple HTTP transactions *share* this connection. The transactions correspond to the retrieval of the individual components of a Web page and perhaps also multiple Web pages.

The logical transfers (or logical data streams) corresponding to the individual components of a Web page are mapped onto this connection. Furthermore, the connection is made *persistent*, i.e., long-lived, so that it can be shared across multiple client-server interactions that are spaced apart in time.

We need a way to map the logical transfers (or logical data streams) corresponding to the individual components of a Web page onto the persistent connection. A straightforward mapping would be to arrange the logical transfers in sequential order. So a typical Web page download would involve a request-response exchange for the HTML file followed, in sequence, by similar

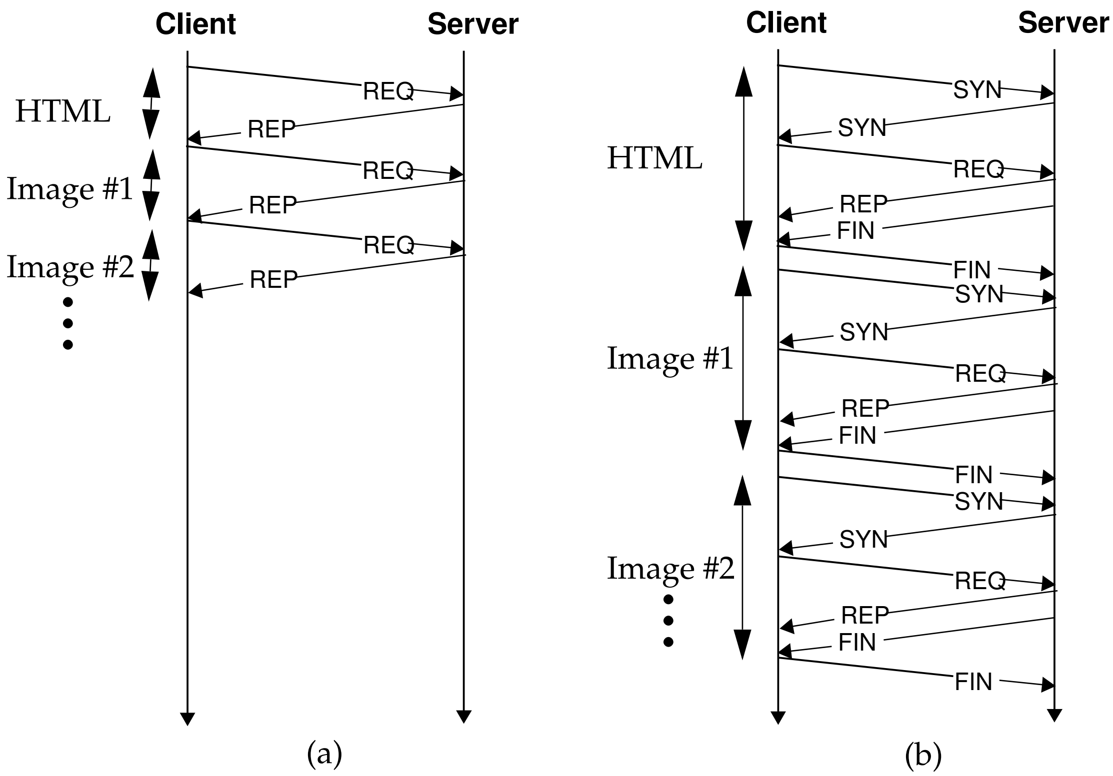


Figure 5.1 Timeline of a Web page download with (a) a persistent connection, and (b) a separate connection for each component. In both cases the request-response exchange for the individual components happen sequentially, one after the other. In case (a) we assume that the persistent connection has already been established by a previous interaction between the client and the server.

exchanges for each inline image. Figure 5.1(a) shows the timeline, assuming that the persistent connection has already been set up by a previous interaction between the same client-server pair. For comparison, we also show the HTTP/1.0 timeline with sequential retrieval in Figure 5.1(b) (reproduced from Figure 4.1(a)).

One clear benefit of a persistent connection is that it amortizes the overhead of TCP connection establishment over multiple logical data transfers, and even multiple Web page downloads. As illustrated in Figure 5.1, the download of a Web page composed of an HTML file and *nimages* inline images would benefit from a reduction of $(1 + nimages) \cdot rtt$ in latency under P-HTTP.

A second benefit of a persistent connection is that the overhead of slow start and other aspects of the TCP congestion avoidance/control algorithm is likewise amortized over multiple logical data transfers. From the viewpoint of the TCP connection, the several distinct logical data streams appear as a single data stream. The logical structure of the sub-stream within the TCP connection has no impact on the TCP algorithms. Thus in Figure 5.1(a), the slow start process initiated during the transfer of the HTML file continues through the subsequent transfer of the inline images. (Of course, if packet loss is detected, the congestion window size would be cut down and the linear growth phase possibly initiated.)

Unfortunately, the benefit of a persistent connection with respect to slow start overhead may not extend to multiple Web page downloads spaced apart in time even if they share the same connection. This occurs because TCP resets the congestion window to its initial value when a connection is idle for a certain period (Section 2.1.5). This phenomenon has been termed the *slow start restart problem* in [44]. The threshold used for the idle period is the RTT or the RTO, both of which tend to be smaller than the user "think time" that spaces successive Web downloads apart. Therefore, when the user requests a new Web page after the think time, slow start is initiated afresh. This increases latency.

Our solution is a new technique that we call *TCP fast start*. To first order, TCP fast start avoids repeated slow start by reusing the congestion window size cached in the recent past. We defer a detailed discussion to Chapter 8.

5.2.1 Multiplexing Logical Data Streams

Recall from Section 2.1 that a TCP connection only provides an ordered byte-stream data delivery service. The protocol does not provide any means for demarcating messages or logical data streams within a connection. Therefore, to multiplex several logical data streams onto a persistent connection, the sender (typically the server in a Web interaction) would, at the least, need a mechanism for demarcating messages at the application level. The server could then send out the messages (logical data streams) sequentially one after the other.

A simple mechanism to demarcate messages that are multiplexed onto a single TCP connection is to add the length of a message as a prefix to the body of the message itself. The *content-length* field in the HTTP/1.0 protocol header could be used for this purpose. One issue to consider, though, is that the length of a message is not always known at the beginning of a transmission, for instance when the message is generated as the output of a server-side script. In such cases, it is difficult to include the content-length field. We consider several alternative solutions:

- *Boundary delimiter*: The server could insert a boundary delimiter (perhaps as simple as a single character) if it can examine the entire data stream and "escape" any instance of the delimiter that appears in the data (as is done in the Telnet protocol [90]). This requires both the server and client to examine each byte of data, which is clearly inefficient.
- *Block-by-block data transmission*: The server could read the output of the script and send it to the client in arbitrary-length blocks, each preceded by a length indicator. A zero-length block would mark the end of a message. This would not require byte-by-byte processing, but it would involve a lot of extra data copying on the server, and would also require a protocol change.
- *Store-and-forward*: This is an extreme case of block-by-block data transmission where the block is the entire output of the script. The server could measure the length of the block and include it in the content-length field. This obviates the need for a protocol change (because the standard content-length field could be used), but requires extra copying and is not practical for arbitrarily large objects.
- *Separate control connection*: The server could use a separate control connection (as in the FTP protocol [91]) to notify the client of the length of a message that it transmits on the data connection. However, the extra connection adds overhead.

Because of the overhead imposed on the server (and possibly the client) by each of these approaches, we chose a simple hybrid approach. In this approach, the server keeps the TCP connection open and uses the content-length field in cases where it can determine the length of the message prior to transmission. In other cases, it marks the end of the message by closing the connection, just as in HTTP/1.0. In the common case (serving out static files), this avoids the cost of extra TCP connections; in the less common case (invoking scripts to generate content dynamically), it may involve the overhead of extra connections but does not add data-touching operations on either the server or the client, and requires no protocol changes. Static files were dominant compared to dynamically-generated data around the time we designed P-HTTP (ca. 1994), and continue to be so. (For instance, [41] reports that fewer than 1% of the bytes and 2% of the files in a large Web client trace were of MIME type CGI (Common Gateway Interface) [18] which represents the majority of dynamically generated data.) Note that a static file refers to one whose size is fixed. It does not necessarily mean static content. In fact, it includes such dynamic content as animated GIF [37,38], dynamic HTML (DHTML) [24], and Java applets [57].

Even dynamically-generated pages contain significant amounts of static data in the form of banners, buttons, text, etc. Therefore, keeping a persistent connection open may be advantageous even for dynamically-generated pages. So the client opens separate non-persistent connections for dynamically-generated data and reserves the persistent connection for static data. The client uses a simple heuristic to distinguish static data from dynamic data: if the HTTP request method is GET and the URL does not contain a "?", the data is assumed to be static and the persistent connection is used. In all other cases, a new non-persistent connection is established. Note that this heuristic is only a performance enhancement and does not impact the correctness of the protocol.

Dynamically-generated may data become more important in the future. Web-based search engines and database front-ends are examples of applications where this is happening. Rather than redesign P-HTTP to avoid the attendant performance penalty, we sidestep the problem by developing an entirely new technique, *TCP session*, which we discuss in Chapter 6.

5.2.2 Negotiating the Use of Persistent Connections

Persistent connections require both servers and clients with capabilities over and beyond those required by HTTP/1.0. Servers must multiplex multiple messages onto a persistent connection while the client must parse the messaging boundaries using the content-length field, and extract the individual messages. Successful deployment requires interoperability between servers/clients that have this capability and those that do not. One way to realize this interoperability is to introduce a mechanism for servers and clients to negotiate the use of persistent connections.

To provide such a negotiation mechanism, we defined a new *hold-connection-open* HTTP *pragma directive* [10] that specifies the use of persistent connections. To negotiate the use of persistent connections, the client includes this directive in its HTTP request. If the server understands the directive, it holds the connection open and echoes the pragma in its response. At this point, both the client and the server have successfully negotiated the use of a persistent connection. If the server does not understand the *hold-connection-open* pragma directive, it simply ignores the directive. So negotiation fails and the client and server revert to the standard HTTP protocol. Similarly, if the server receives a request that does not include the directive, it assumes that the client does not support persistent connections and falls back to the standard protocol.

Subsequent to our work, the HTTP/1.1 protocol has defined a similar negotiation mechanism as part of the protocol. However, HTTP/1.1 is different from P-HTTP in one respect, and that is it designates the use of a persistent connection as the *default* behavior of clients and servers.

5.2.3 Terminating Persistent Connections

A persistent TCP connection consumes resources both at the server and the client. These resources include the TCP protocol control block and the socket buffer. Because of resource constraints, the server and/or client may wish to limit the number of persistent connections it holds open at any one time. When the number of persistent connections is at the limit and a new connection is needed, (at least) one of the existing connections must be terminated. To solve this problem, we use a simple least-recently-used (LRU) policy to pick the victim. It is possible to use more sophisticated policies, such as picking victims in increasing order of the round trip time, but we did not investigate this issue for this thesis.

The hosts at either end of a persistent connection must handle the case that their peer closes the persistent connection without notice. The closure of the connection could result in several failures from the viewpoint of a typical client-server interaction:

1. A client's attempt to send a new request may fail.
2. The client may succeed in sending its request but the server may not succeed in sending its reply.

A failure of the first kind, while undesirable, can be dealt with by having the client establish a new (persistent) connection and send its request again. However, a failure of the second kind can have more serious consequences because the server would have processed the client's request but the client would not know about it. Although many HTTP requests do not have side-effects and most are idempotent, some are not. (An example of one that is not idempotent is a request to cast a vote in a Web-based on-line poll.) In the latter case, there is no graceful way of recovering from a failure.

Therefore, our policy is to prevent the client or the server from closing a connection in the midst of a request-response exchange. This is easy to do because each can determine, using only local knowledge, whether there are any requests or responses outstanding. This avoids the second failure mode, so only the first remains. There are two ways in which the first failure mode can occur:

- The server has already informed the client host that it has closed the connection. In this case, the client program may also be informed of this before it tries to reuse the connection for a new request to the same server. At worst, it will find out when its attempt to send out the request fails immediately because the network stack is aware that the connection has been closed.
- The server's message indicating connection closure (TCP FIN packet) and the client's request cross each other on the network. In this case, the server rejects the request when it receives it. The client program discovers the failure when its attempt to read the reply fails.

In either case, the client program recovers from the failure by establishing a new connection and sending the request afresh.

5.3 Pipelining

Even with a persistent TCP connection, a simple implementation of the HTTP protocol would still require at least one network round trip to retrieve each inline image, as illustrated in Figure 5.1(a). The client interacts with the server in a stop-and-wait fashion, sending a request for an inline image only after having received the data for the previous one.

However, this lockstep sequence of requests can be avoided since one component of a Web page (such as an inline image) in no way depends on previous components. The client could, therefore, *pipeline* multiple requests over the same connection. We consider two ways in which the client requests could do this: GETALL and GETLIST. We note at the outset that while pipelining could be applied in the context of Web page components of arbitrary type, our description here assumes inline images, the most common type of component embedded in Web pages.

5.3.1 The GETALL method

When a client does a GET on a URL corresponding to an HTML document, the server just sends back the contents of the corresponding file. The client then sends separate requests for each inline

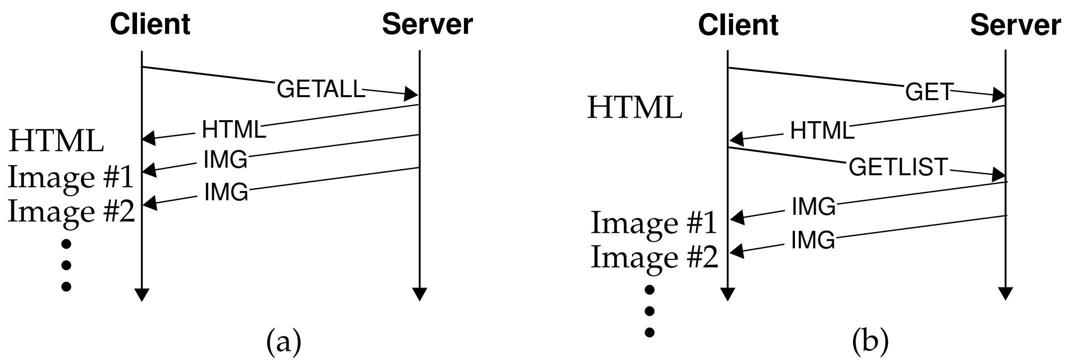


Figure 5.2 Timeline of a Web page download over a persistent connection with pipelining enabled. (a) shows the use of GETALL, and (b) shows the use of GETLIST.

image. Typically, however, most or all of the inline images reside on the same server as the HTML document, and will ultimately come from the same server.

We extended HTTP with a new *GETALL* method, which specifies that the server should return an HTML document and all of its inline images that reside on that server. On receiving this request, the server parses the HTML file to find the URLs of the images, then sends back the file and the images in a single response. The client uses the content-length field in the individual HTTP headers to split the response into its components (as described in Section 5.2.1).

Because GETALL relies on the server to parse HTML files, it imposes a new burden on the server. However, this additional overhead is offset by the server being relieved from parsing many additional HTTP requests, which the standard GET method would entail. If the cost of parsing HTML files is deemed too high, the server could keep a cache of the URLs associated with a subset of heavily-accessed HTML files, or even all of them.

The GETALL method can be implemented using the standard GET method, with a special pragma directive included in the request header to indicate that the client wants to perform a GETALL. This allows the client to inter-operate with a server that does not support GETALL. If the server supports GETALL, it includes the GETALL pragma directive in its reply header. The client waits for all the HTML and all the inline images. If the server does not support GETALL, it simply ignores the pragma directive. The non-inclusion of this directive in the server's response indicates to the client that it should revert to the standard GET method.

Web clients typically maintain a local cache of Web page components, such as inline images, to avoid unnecessary network interactions. However, a server has no way of knowing which of the inline images in a document are in the client's cache. So the GETALL method causes the server to return all the inline images, thereby defeating client-side caching in general. GETALL would still be useful in situations where the client knows that it has no relevant images cached (for example, if its cache contains no images from the server in question, or local caching is turned off).

5.3.2 The GETLIST method

To get around the limitation of the GETALL method, we defined a new *GETLIST* method, which allows a client to request a set of documents or images from a server. A client can use the GET method to retrieve an HTML file, and then use the GETLIST method to retrieve, in one exchange, all the images not in its cache.

Logically, a GETLIST is the same as a series of GETs sent back-to-back. In fact, we chose to implement it this way, since it requires no protocol change and it performs about the same as an explicit GETLIST would.

The GETLIST method incurs an additional network round trip (to retrieve the HTML file) compared to GETALL. However, more significantly, it avoids the drawback that GETALL with regard to client-side caching. So GETLIST is our method of choice for pipelining HTTP requests.

5.4 Implementation

We briefly describe two different implementations of P-HTTP that we have done. The first, involving a modified client and server, was our original one and is described in [85]. The second, our proxy-based implementation, is more recent.

5.4.1 Client-Server Implementation

The two components of P-HTTP, persistent connections and pipelining, require support at both the client and the server. Specifically, the use of persistent connections involves the following actions on the part of the client and the server:

- The client indicates its desire to use a persistent connection by including a *hold-connection* pragma in its initial request. It uses the content-length field to indicate the length of the request it is sending.

- If the server is capable and willing to use a persistent connection, it includes the *hold-connection* pragma in its response. It uses the content-length field in the request header to determine when it has received the entire request. Whenever possible, the server marks the end of its response using the content-length field. It closes the connection to a client when the content-length is not known beforehand. It also closes connections to conserve resources — either when the number of open connections exceeds a threshold or when a connection has been idle for longer than a threshold. Both these thresholds can be configured as appropriate.

- The client checks to see if the server's response includes the hold-connection pragma. If it does and if there is a content-length field in the reply, the client reads in the corresponding amount of data from the connection. It also uses the same (persistent) connection to send requests to the server in the future. If the server's response does not include either the hold-connection pragma or the content-length field, the client reads in data until the connection is closed by the server. Also, it establishes a new connection for future requests.

- If the client encounters the closure of the persistent connection by the server, either when trying to send a new request or while waiting for a response, it establishes a new connection and retries the request. If the server encounters a connection closure by the client, it does nothing. Since HTTP interactions are always initiated by the client, the onus of re-establishing the connection, if necessary, is placed on the client.

In addition, the client and the server do the following to support pipelining:

- Once the server has confirmed that it supports persistent connections (for instance, as part of the exchange to obtain the HTML file), the client is free to pipeline requests. It sends requests back-to-back using the content-length field in each request header to indicate its length.

- The server processes the requests from the client sequentially and sends back its response to each in the same sequence. It does not interleave the responses.

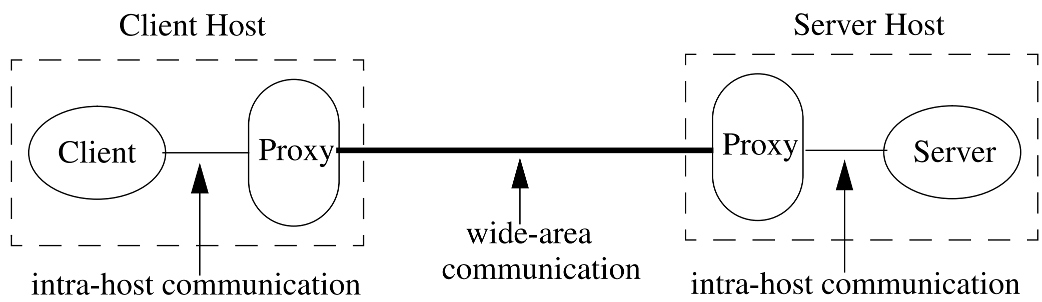


Figure 5.3 The architecture of a proxy-based implementation of P-HTTP. P-HTTP operates only between the two proxies, leaving the client and server programs untouched.

- It is essential that the server use the content-length field to indicate the end of a response. If it closes the connection instead, the client will not receive responses to later requests it may have pipelined. The client uses the same heuristic as mentioned in Section 5.2.1 to avoid forcing the server to close the connection because of dynamically-generated data. It only pipelines requests that use the GET method and do not include a '?' in the URL. A '?' indicates the likely use of a server-side script which may force the server to close the connection prematurely. Despite the restrictions, the client is able to pipeline the majority of requests, which are for static[2] files.

We implemented the client and server actions listed above by modifying the *Mosaic* version 2.4 browser and the *httpd* version 1.3 server, both from NCSA [77]. These represented the leading-edge in publicly-available client and server software at the time we did the implementation (circa mid-1994). We ported the modified software to two platforms: DEC MIPS running Ultrix and DEC Alpha running OSF. The modified software has been available via anonymous FTP since 1995 [86].

5.4.2 Proxy-based Implementation

Although our initial implementation of P-HTTP influenced later developments, including HTTP/1.1, it has the drawback of being tied to specific client and server software. As NCSA Mosaic and httpd have been replaced by newer software with added features, the reach of our implementation

2. As noted in Section 5.2.1, "static" files encompass "dynamic" content such as animated GIFs and Java applets.

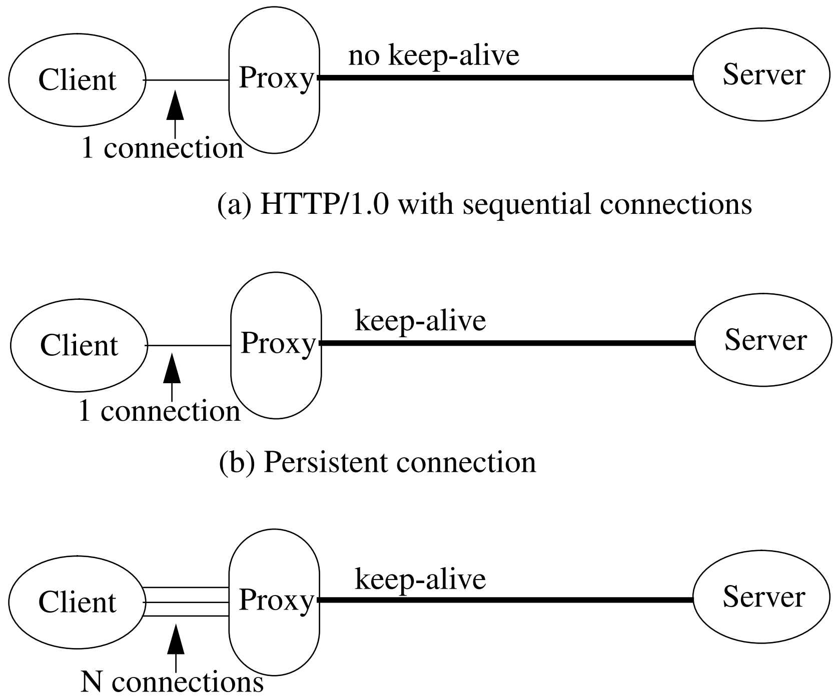
has been restricted. Its suitability for performance testing has diminished because newer client and server software (such as Netscape Navigator [78], Internet Explorer [72] and Apache httpd [1]) include optimizations that are orthogonal to P-HTTP.

To get around these problems, we have implemented a simple, proxy-based version of P-HTTP. The key observation is that P-HTTP is primarily a technique to optimize network communication between a client and a server. If the client and server applications communicate with each other directly, implementing P-HTTP would involve modifying the applications. The key to avoiding this is to have the client and server communicate *indirectly* via proxies (Figure 5.3). With the proxies co-located with the client and the server, the network communication between the two proxies is the primary determinant of performance. That between the client and its proxy, and likewise the server and its proxy, constitutes intra-host or intra-LAN communication, and as such is far less expensive. Therefore, we can achieve much of the benefit of P-HTTP by having the proxies use a persistent connection and pipeline requests and responses on this connection.

The server software we used, Apache httpd version 1.3b5, supports the HTTP *keep-alive* mechanism which is similar to the *hold-connection* mechanism in our original implementation. Further, the server is capable of using the content-length field to separate out multiple requests and responses on the same connection. Therefore, we did not require a proxy at the server end.

The client software we used, Netscape Navigator version 3.01, allows us to configure the maximum number of concurrent connections that are opened to the same server. When a (local) proxy is used, this controls the number of concurrent connections established between the browser and the proxy. By varying the number of connections and by having the client-side proxy choose whether or not to use the keep-alive mechanism, we emulate there different protocol configurations:

1. *HTTP/1.0 with sequential connections* (Figure 5.4(a)): Only one connection is established at a time between the browser and the proxy, and the proxy does not use the keep-alive mechanism when communicating with the server. The net result is that a new connection is established between the proxy and the server for each HTTP request-response interaction.



(a) HTTP/1.0 with sequential connections

(b) Persistent connection

(c) Persistent connection with pipelining (P-HTTP)

Figure 5.4 Using a proxy-based implementation to emulate various protocol combinations.

2. *Persistent connection* (Figure 5.4(b)): Only one connection is established at a time between the browser and the proxy, but the proxy does use the keep-alive mechanism when communicating with the server. Consequently, a persistent connection is established and used for communication between the proxy and the server. However, the single connection between the browser and the proxy prevents the pipelining of requests and responses.

3. *Persistent connections with pipelining* (P-HTTP) (Figure 5.4(c)): Up to N (>1) connections are established at a time between the browser and the proxy, and the proxy uses the keep-alive mechanism when communicating with the server. As a consequence, a persistent connection is established and used for communication between the proxy and the server. Moreover, the proxy pipelines up to N requests on the connection.

While the proxy-based implementation of P-HTTP is certainly more generally applicable than our original implementation, we decided against using it for our experiments. Our reasons for this decision are discussed in the next section.

5.5 Experimental Results

For our experiments, we wrote a simple Web client program that emulated HTTP/1.0, persistent connections and pipelining. This program sends out requests and reads in the responses, but does not display the retrieved material. We chose to use this rather than our proxy-based implementation because:

1. We discovered via experiments that the Netscape Navigator version 3.01 browser does not allow launching more than 6 concurrent connections. This limits the maximum number of requests that can be pipelined to 6, which is clearly undesirable.
2. A human user is needed to operate the browser, making it difficult to automate the experiment.
3. Using the simple client program allows us to isolate the network performance from other orthogonal issues such as rendering speed.

We conduct experiments using two different network configurations.

1. *Terrestrial wide-area network* (Figure 5.5(a)): The client host is located in Berkeley, CA, USA and the server host in Germantown, MD, USA. A 13-hop terrestrial path connects the server to the client and a 14-hop terrestrial path provides connectivity in the reverse direction. The client and server are connected to 10 Mbps Ethernet segments. The round trip time between the client and server is approximately 70 ms.
2. *Satellite-based wide-area network* (Figure 5.5(a)): The client and server locations are the same as above, but the client host also has an interface connected to the DirecPC satellite network [27]. Client-to-server communication happens via the same 13-hop terrestrial path as before, but communication in the reverse direction happens via a 2-hop path that includes a geostationary satellite link. The round trip time between the client and server via these asymmetric paths is approximately 335 ms.

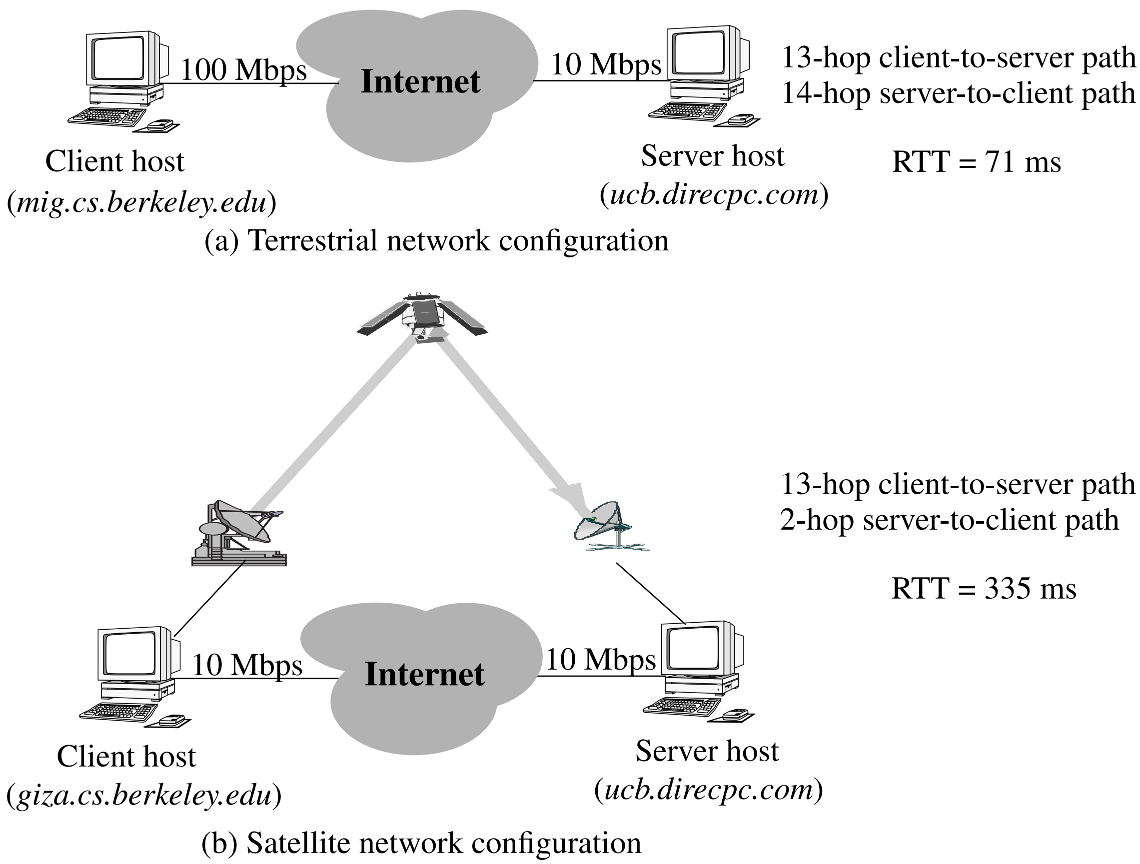


Figure 5.5 The two network configurations used for our experiments. Note that both the browser and the proxy run on the client host.

We create several test Web pages on the server. Each test page contains 1-10 inline images of identical sizes. We use GIF images of 3 sizes: 250 bytes (representative of small buttons or icons), 5 KB (approximately the average size of a GIF image as reported in [41]), and 40 KB (representative of larger images such as photographs). Thus we have a total of 10x3 = 30 test pages. We download each test page from the server and record the time taken, repeating this experiment 20 times. We turn off local caching at the browser to force the HTML file and the inline images to be downloaded from the server each time. We report the average download time, together with 95% confidence intervals, for each combination of image size and image count.

5.5.1 Terrestrial Wide-Area Network

Figure 5.6 shows the performance results for the experiments conducted across the terrestrial wide-area network. The general observation is that the use of persistent connections improves performance substantially. Pipelining results in an additional improvement.

In relative terms, the improvement in performance is greatest when the image size is the smallest. For instance, with 10 inline images, persistent connections and pipelining together decrease the average download time by approximately a factor of 6.13 for 250-byte images, 5.67 for 5 KB images and 4.73 for 40KB images. The reason is that the smaller the image size, the greater the cost of connection establishment and slow start as a fraction of the total download time. Since the use of persistent connections and pipelining reduces these costs, there is greater improvement when these costs are large in relative terms. We observe a similar trend when comparing the benefit of persistent connections and pipelining to that of persistent connections alone.

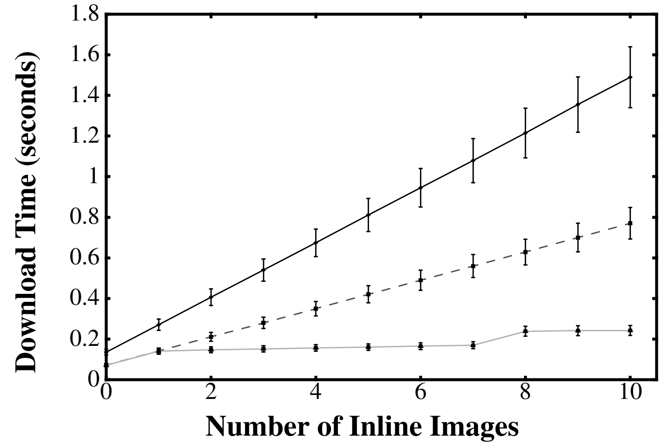
In absolute terms, however, the performance improvement is greater when the inline images are larger in size and number. This is evident from the different slopes of the curves in Figure 5.6, both within the individual graphs and across the three graphs. The reason for the different slopes is that the cost downloading an individual image is different in each case. With HTTP/1.0, it includes the cost of connection setup, the round trip consumed to request the image and start receiving the reply, and the entire cost of slow start for transmitting the reply. With persistent connections, the connection setup and slow start costs are amortized over several individual images, but the round trip for request-response exchange for each image remains. With pipelining added on, the round trip for the request-response exchange for the individual images overlap, further decreasing the cost per image.

5.5.2 Satellite-based Wide-Area Network

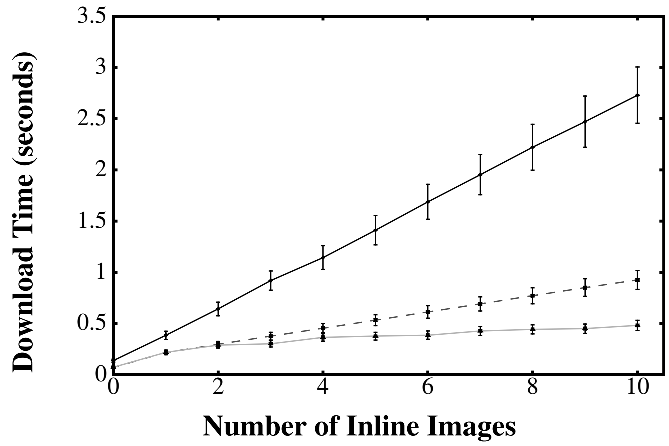
Figure 5.7 shows the performance results for the experiments conducted over the DirecPC satellite network. The general performance trends are the same as for the case of the terrestrial network. However, the performance gains in absolute terms is much larger than in the case of the terrestrial network owing to the much larger round trip time in the former.

Legend: —— HTTP/1.0
- - - - Persistent connection
· · · · · · Persistent connection + pipelining (P-HTTP)

(a) Image size = 250 bytes



(b) Image size = 5 KB



(a) Image size = 40 KB

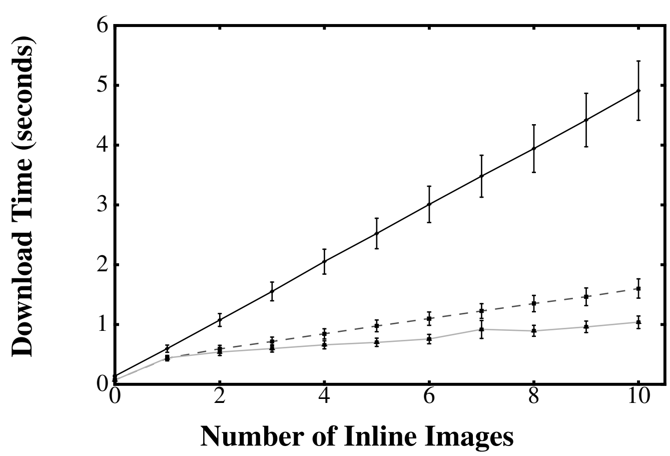


Figure 5.6 Results of experiments conducted across a terrestrial WAN. The error bars correspond to 95% confidence intervals for the mean.

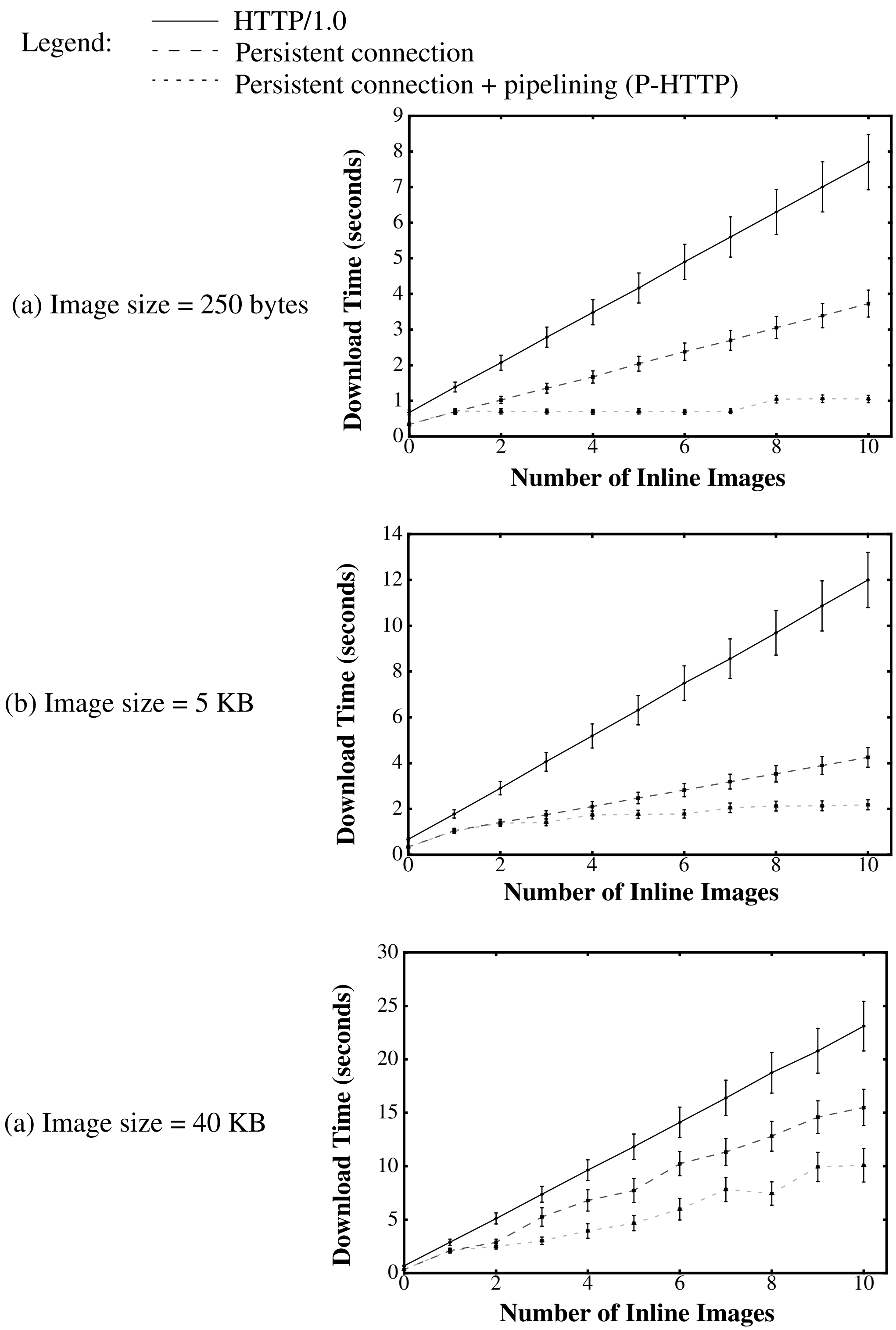


Figure 5.7 Results of experiments conducted over the DirecPC satellite network. The error bars correspond to 95% confidence intervals for the mean.

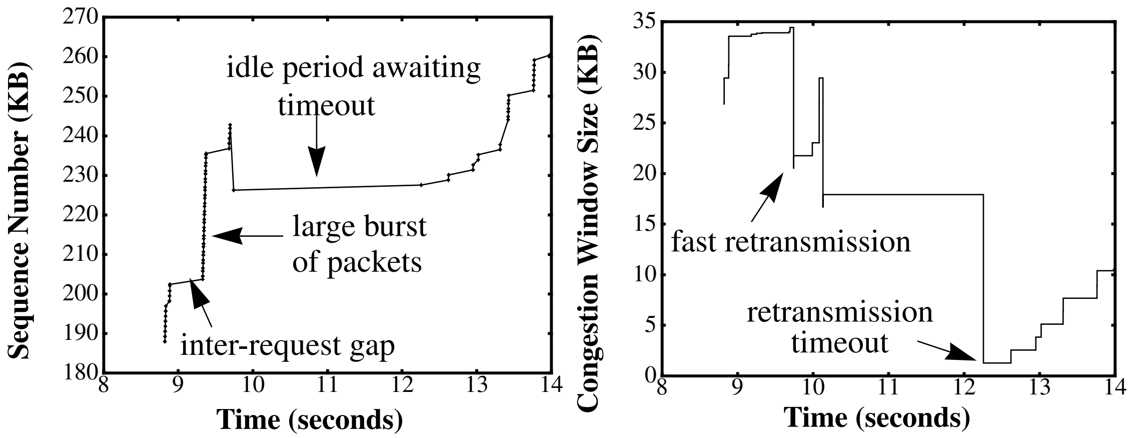


Figure 5.8 A section of the sequence number and congestion window size traces illustrating the problem that arise when the download of a new inline image begins on a persistent connection but pipelining is not in use. The inline image size was 40 KB and the experiment was conducted over the DirecPC network.

5.5.3 Interaction of P-HTTP with TCP Congestion Control

We now discuss two ways in which persistent connections and pipelining interact with the TCP congestion control algorithm. The first interaction arises when a persistent connection is used to download a Web page with several inline images, but the requests and responses are *not* pipelined. In this case, there is a period of duration approximately one RTT between successive image downloads during which the server idles waiting for the client to send its next request. Ack clocking dies down during this period because the server has no more data to send pending the client's new request. However, the server's congestion window size for the persistent TCP connection remains unchanged (because the idle period is not long enough for the congestion window size to be reset as discussed in Section 2.1.5). So when a new request arrives, the server starts off by sending a large burst of packets back-to-back. This could lead to heavy packet loss, often forcing the server to suffer a retransmission timeout. Figure 5.8 illustrates this using a sequence number trace and a corresponding congestion window size trace. Note that the use of pipelining avoids this problem by eliminating the gap between the download of successive images.

The second interaction with the TCP congestion control algorithm arises when the persistent connection remains idle between successive Web page downloads for long enough that the server resets its congestion window size. Such pauses are common during a Web browsing session as the

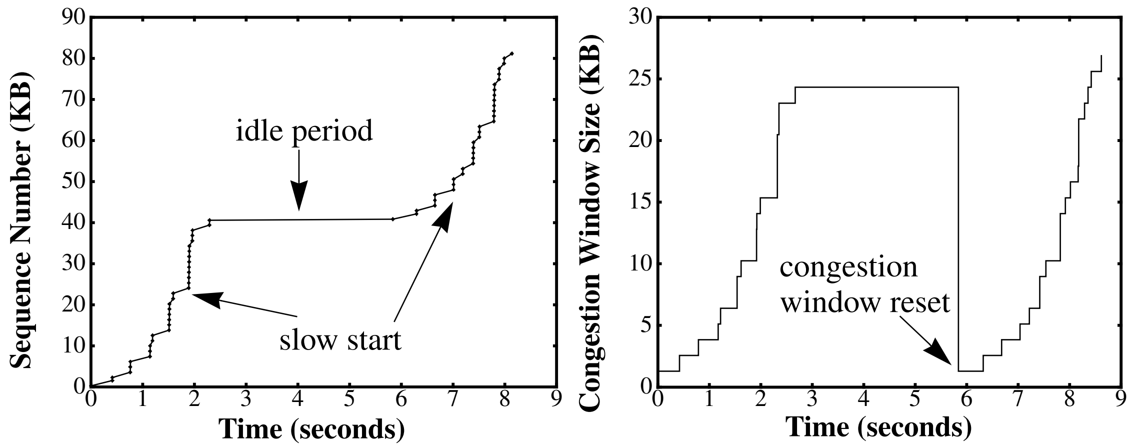


Figure 5.9 Sequence number and congestion window plots illustrating the problem of repeated slow start when two downloads over a persistent connection are spaced apart in time. This experiment was conducted over the DirecPC satellite network.

user usually spends some time perusing the retrieved material before initiating the next download. As a consequence, the entire penalty of slow start is incurred when the connection resumes activity, presumably to download a new Web page. This problem has been reported in the literature (for example, in [44], where it is called the *slow start re-start problem*), and Figure 5.9 illustrates it using traces. In Chapter 8, we present *TCP fast start,* our solution to this problem. TCP fast start tries to avoid repeated slow start by caching and reusing the congestion window size, but at the same time taking care to avoid a penalty when the cached information is stale.

We now turn to other work that has adopted and extended the key ideas of persistent connections and pipelining in P-HTTP.

5.6 Extensions of P-HTTP

We discuss the use of persistent connections in HTTP/1.1 and alternative ways of multiplexing data within a persistent connection.

5.6.1 Persistent Connections in HTTP/1.1

Our P-HTTP work has had a significant impact on the new version of the HTTP protocol, HTTP/1.1 [30], which was standardized in 1997. Our work is cited in the standard document, RFC 2068. Among the various new features and optimizations included in HTTP/1.1 are persistent connections and pipelining, as suggested by our work. The use of persistent connections, while not mandatory, is prescribed as the default behavior, i.e., clients and servers can assume that their peer supports persistent connections unless explicitly notified otherwise. A chunked encoding scheme, like the block-by-block transmission scheme described in Section 5.2.1, is used to when the content-length is not known. The standard allows clients to have up to 2 persistent connections open to the server, instead of just 1. The rationale is to decrease the chance of having a short transfer blocked by a previous long transfer. We discuss this head-of-the-line blocking problem further in Section 5.7, as a lead in to our work on TCP sessions. A performance analysis of HTTP/1.1 appears in [80].

5.6.2 Application-independent Multiplexing Protocols

P-HTTP multiplexes several logical data streams onto a TCP connection. But this is done with the active involvement of the client and server applications. An alternative would be to abstract out the multiplexing functionality into a separate library, such as the socket library that interfaces applications to the TCP/IP protocol stack. Session Control Protocol (SCP) [100] and the Session Multiplexing Protocol (WebMUX) [36] are two examples of such an approach. These protocols support more fine-grained interleaving of data belonging to different logical streams than either P-HTTP or HTTP/1.1. However, they suffer from some of the same drawbacks as P-HTTP and HTTP/1.1. First, applications would still have to be modified, or at least re-linked, to work with the new socket library. Second, the head-of-line blocking problem, alluded to in Section 5.6.1, still persists. We discuss these in more detail next.

5.7 Limitations of P-HTTP

In spite of the performance benefits, P-HTTP has its limitations. Some of these arise from the interaction with the TCP congestion control algorithm, while others arise because of the restrictiveness of the TCP service model. Many of the limitations apply equally to alternative application-level multiplexing techniques such as MUX and SCP discussed in Section 5.6.2.

1. The reliable, ordered byte-stream service provided by TCP causes logically-independent data streams that are multiplexed onto a single TCP connection to get coupled together in undesirable ways. For instance, data belonging to a certain inline image would get held up whenever data belonging to a different image, but which happens to lie earlier in the sequence number space, gets lost in the network and is awaiting retransmission[3]. Another consequence of the coupling is that it is difficult or even impossible to treat individual Web objects (such as inline images) differently in the interior of the network. Having such a capability would be useful, for instance, to selectively intercept certain Web requests and responses at a transparent cache (e.g., [19]) but let the others go through without interference.

   The approach adopted by HTTP/1.1 to address this problem is to allow the setting up of two persistent connections between the client and the server rather than just one. Clearly, this does not really solve the problem because the number of logical data streams could well exceed two. Furthermore, increasing the number of connections is counter to the basic goal of sharing and reusing connections.

2. The coupling of logically separate data streams can lead to denial-of-service attacks, as noted in [35]. A client could connect to a server via a proxy and start downloading a large file. This would tie up the persistent connection between the proxy and the server for a long time, effectively shutting out other clients that wish to connect to the same server via the proxy.

3. Sharing a connection between multiple data streams requires either that the application do the multiplexing explicitly or that it use a special communications library that does the multiplexing. These require the application to be modified, or at the least relinked with the new library. Since both the communicating peers (i.e., client and server in the context of the Web) have to agree on the framing format for multiplexing, modifications would be required at both ends.

---

3. This is akin to the *head-of-line blocking* problem in input-buffered switches.

Such modifications are undesirable when one considers that the need to support short and bursty data streams efficiently could extend beyond the Web, in particular to legacy applications, such as distributed database transaction processing, that may be difficult or expensive to modify.

4. It is difficult or impossible for separate application processes to share a single TCP connection. Therefore a Web browser and a helper application, both of which wish to communicate directly with a server, would end up using separate connections.

5. As mentioned in Section 5.5.3, the use of a persistent connection does not avoid the invocation of slow start each time the connection resumes activity after a pause. Since the total size of a Web page, including all embedded components, itself tends to be small on average (typically 20-30 KB), the cost of the repeated slow start can be quite significant.

5.8 Summary

In this chapter, we have discussed persistent-connection HTTP (P-HTTP), our initial solution to the performance problems that afflict HTTP/1.0. The two main ideas underlying P-HTTP are persistent connections and pipelining. The use of persistent connections amortizes the cost of connection set up over multiple Web page downloads. It also amortizes the slow start penalty over the download of multiple components (such as inline images) that constitute individual Web pages. Pipelining requests and responses over the persistent connection helps transfer the components of a Web page back-to-back, without any pauses in between, saving at least one RTT per component. Results from experiments both over a terrestrial wide-area network as well as a satellite-based one show that P-HTTP results in a significant improvement in performance, with a reduction in download time by up to a factor of 7.

The primary drawback of P-HTTP is that it couples together data streams that are otherwise independent. It is interesting to note that using a separate TCP connection for each logical data stream would avoid this problem. However, as discussed in Chapter 4, such an approach can have an adverse effect on the performance of the network as a whole. In the next chapter, we present our new solution, called *TCP session*, which uses a TCP separate connection for each logical data stream but yet does not suffer from the same performance drawbacks.