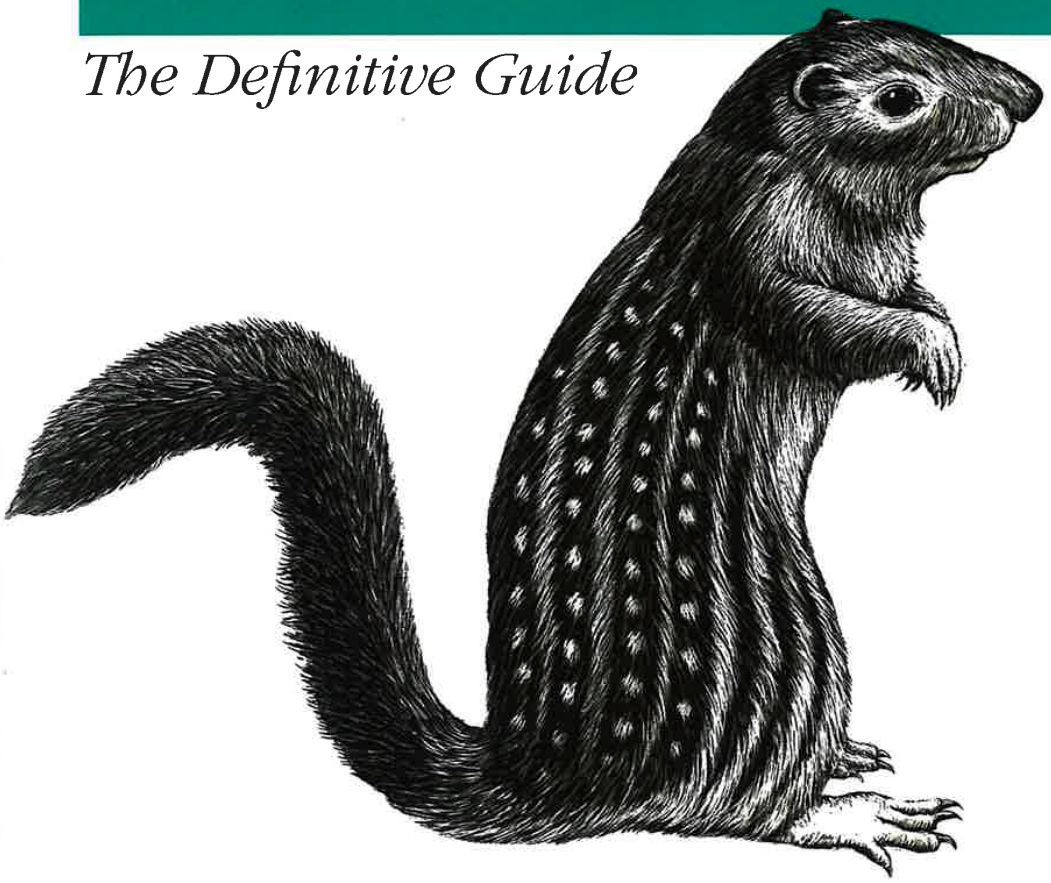
Understanding Web Internals

HTTP

The Definitive Guide



O'REILLY®

David Gourley & Brian Totty
with Marjorie Sayer, Sailu Reddy & Anshu Aggarwal

HTTP

The Definitive Guide

David Gourley and Brian Totty
with Marjorie Sayer, Sailu Reddy, and Anshu Aggarwal

O'REILLY®
Beijing • Boston • Farnham • Sebastopol • Tokyo

HTTP: The Definitive Guide
by David Gourley and Brian Totty
with Marjorie Sayer, Sailu Reddy, and Anshu Aggarwal

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media, Inc. books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor:	Linda Mui
Production Editor:	Rachel Wheeler
Cover Designer:	Ellie Volckhausen
Interior Designers:	David Futato and Melanie Wang

Printing History:

September 2002: First Edition.

ISBN-10: 1-56592-509-2
ISBN-13: 978-1-56592-509-0
[LSI]

[03/2019]

CHAPTER 4
Connection Management

The HTTP specifications explain HTTP messages fairly well, but they don't talk much about HTTP connections, the critical plumbing that HTTP messages flow through. If you're a programmer writing HTTP applications, you need to understand the ins and outs of HTTP connections and how to use them.

HTTP connection management has been a bit of a black art, learned as much from experimentation and apprenticeship as from published literature. In this chapter, you'll learn about:

- How HTTP uses TCP connections
- Delays, bottlenecks and clogs in TCP connections
- HTTP optimizations, including parallel, keep-alive, and pipelined connections
- Dos and don'ts for managing connections

TCP Connections

Just about all of the world's HTTP communication is carried over TCP/IP, a popular layered set of packet-switched network protocols spoken by computers and network devices around the globe. A client application can open a TCP/IP connection to a server application, running just about anywhere in the world. Once the connection is established, messages exchanged between the client's and server's computers will never be lost, damaged, or received out of order.*

Say you want the latest power tools price list from Joe's Hardware store:

http://www.joes-hardware.com:80/power-tools.html

When given this URL, your browser performs the steps shown in Figure 4-1. In Steps 1–3, the IP address and port number of the server are pulled from the URL. A TCP

* Though messages won't be lost or corrupted, communication between client and server can be severed if a computer or network breaks. In this case, the client and server are notified of the communication breakdown.

connection is made to the web server in Step 4, and a request message is sent across the connection in Step 5. The response is read in Step 6, and the connection is closed in Step 7.

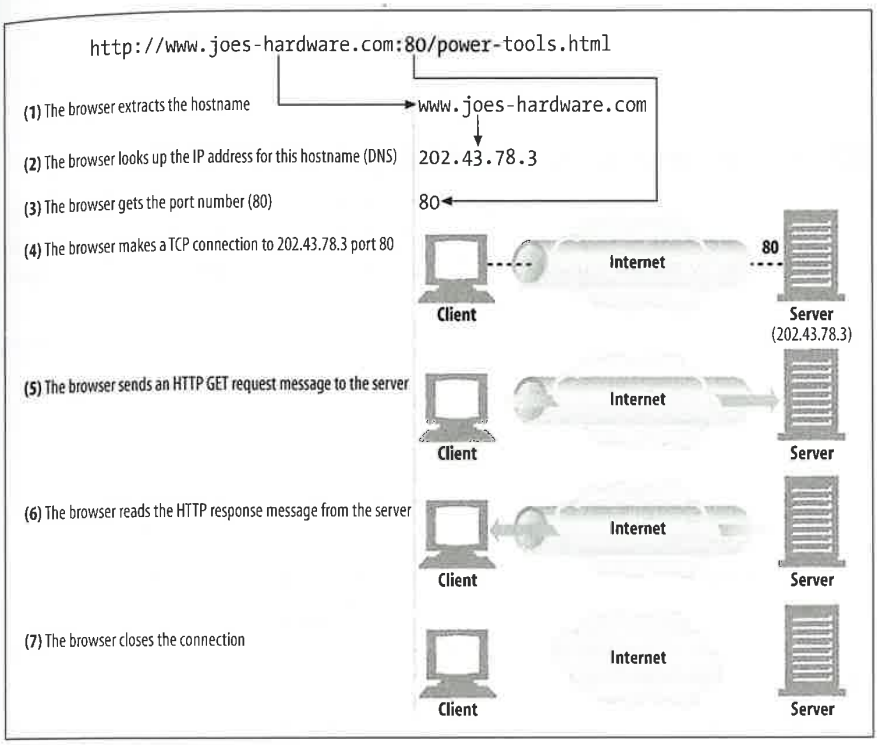


Figure 4-1. Web browsers talk to web servers over TCP connections

TCP Reliable Data Pipes

HTTP connections really are nothing more than TCP connections, plus a few rules about how to use them. TCP connections are the reliable connections of the Internet. To send data accurately and quickly, you need to know the basics of TCP.*

TCP gives HTTP a *reliable bit pipe*. Bytes stuffed in one side of a TCP connection come out the other side correctly, and in the right order (see Figure 4-2).

* If you are trying to write sophisticated HTTP applications, and especially if you want them to be fast, you'll want to learn a lot more about the internals and performance of TCP than we discuss in this chapter. We recommend the "TCP/IP Illustrated" books by W. Richard Stevens (Addison Wesley).

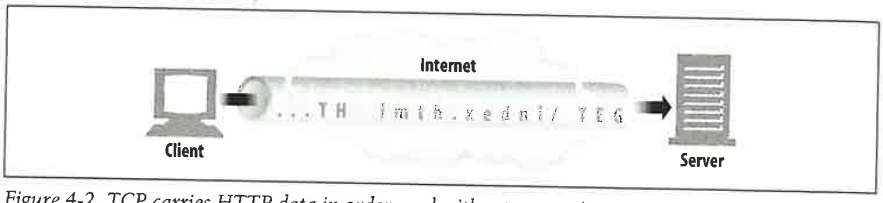


Figure 4-2. TCP carries HTTP data in order, and without corruption

TCP Streams Are Segmented and Shipped by IP Packets

TCP sends its data in little chunks called *IP packets* (or *IP datagrams*). In this way, HTTP is the top layer in a "protocol stack" of "HTTP over TCP over IP," as depicted in Figure 4-3a. A secure variant, HTTPS, inserts a cryptographic encryption layer (called TLS or SSL) between HTTP and TCP (Figure 4-3b).

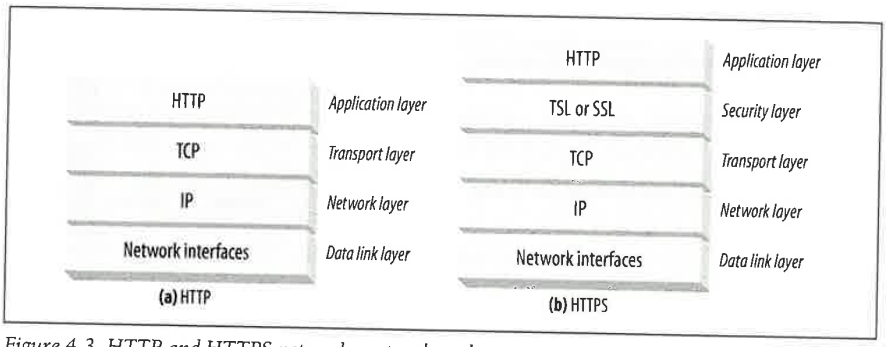


Figure 4-3. HTTP and HTTPS network protocol stacks

When HTTP wants to transmit a message, it streams the contents of the message data, in order, through an open TCP connection. TCP takes the stream of data, chops up the data stream into chunks called segments, and transports the segments across the Internet inside envelopes called IP packets (see Figure 4-4). This is all handled by the TCP/IP software; the HTTP programmer sees none of it.

Each TCP segment is carried by an IP packet from one IP address to another IP address. Each of these IP packets contains:

- An IP packet header (usually 20 bytes)
- A TCP segment header (usually 20 bytes)
- A chunk of TCP data (0 or more bytes)

The IP header contains the source and destination IP addresses, the size, and other flags. The TCP segment header contains TCP port numbers, TCP control flags, and numeric values used for data ordering and integrity checking.

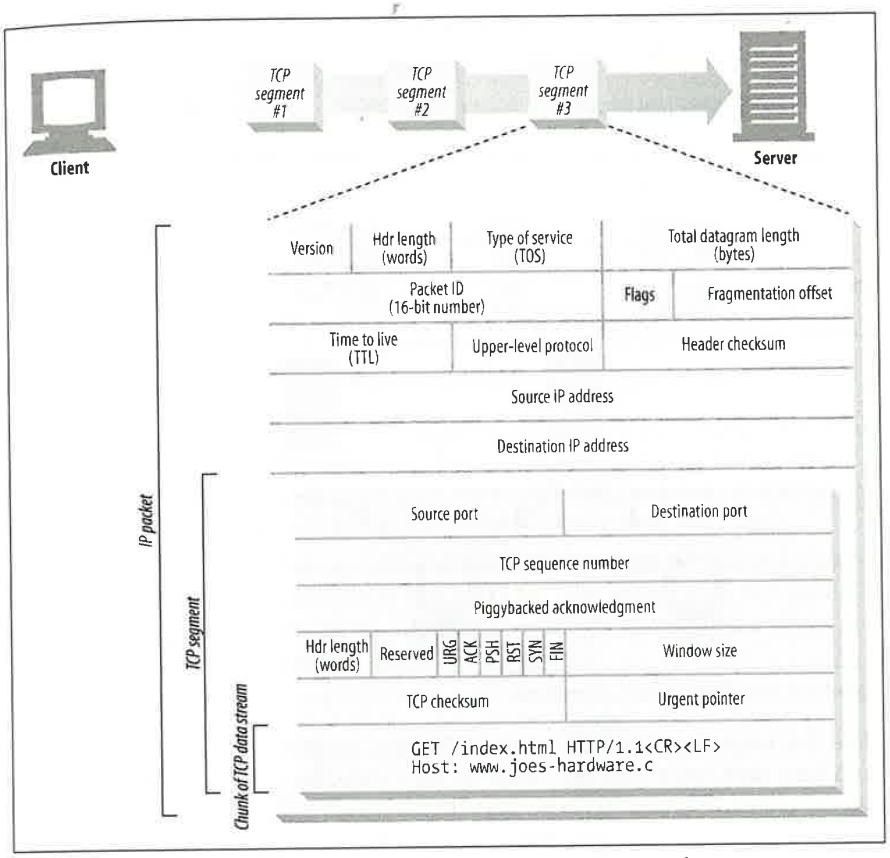


Figure 4-4. IP packets carry TCP segments, which carry chunks of the TCP data stream

Keeping TCP Connections Straight

A computer might have several TCP connections open at any one time. TCP keeps all these connections straight through *port numbers*.

Port numbers are like employees' phone extensions. Just as a company's main phone number gets you to the front desk and the extension gets you to the right employee, the IP address gets you to the right computer and the port number gets you to the right application. A TCP connection is distinguished by four values:

```
<source-IP-address, source-port, destination-IP-address, destination-port>
```

Together, these four values uniquely define a connection. Two different TCP connections are not allowed to have the same values for all four address components (but different connections can have the same values for some of the components).

In Figure 4-5, there are four connections: A, B, C and D. The relevant information for each port is listed in Table 4-1.

Table 4-1. TCP connection values

Connection	Source IP address	Source port	Destination IP address	Destination port
A	209.1.32.34	2034	204.62.128.58	4133
B	209.1.32.35	3227	204.62.128.58	4140
C	209.1.32.35	3105	207.25.71.25	80
D	209.1.33.89	5100	207.25.71.25	80

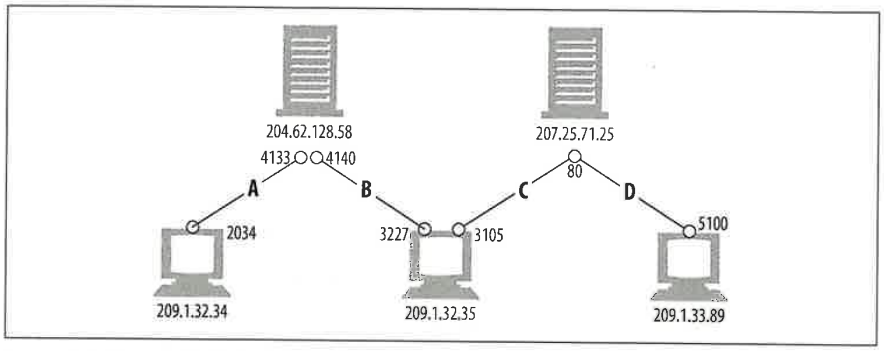


Figure 4-5. Four distinct TCP connections

Note that some of the connections share the same destination port number (C and D both have destination port 80). Some of the connections have the same source IP address (B and C). Some have the same destination IP address (A and B, and C and D). But no two different connections share all four identical values.

Programming with TCP Sockets

Operating systems provide different facilities for manipulating their TCP connections. Let's take a quick look at one TCP programming interface, to make things concrete. Table 4-2 shows some of the primary interfaces provided by the sockets API. This sockets API hides all the details of TCP and IP from the HTTP programmer. The sockets API was first developed for the Unix operating system, but variants are now available for almost every operating system and language.

Table 4-2. Common socket interface functions for programming TCP connections

Sockets API call	Description
s = socket(<parameters>)	Creates a new, unnamed, unattached socket.
bind(s, <local IP:port>)	Assigns a local port number and interface to the socket.

Table 4-2. Common socket interface functions for programming TCP connections (continued)

Sockets API call	Description
connect(s, <remote IP:port>)	Establishes a TCP connection to a local socket and a remote host and port.
listen(s,...)	Marks a local socket as legal to accept connections.
s2 = accept(s)	Waits for someone to establish a connection to a local port.
n = read(s,buffer,n)	Tries to read n bytes from the socket into the buffer.
n = write(s,buffer,n)	Tries to write n bytes from the buffer into the socket.
close(s)	Completely closes the TCP connection.
shutdown(s,<side>)	Closes just the input or the output of the TCP connection.
getsockopt(s, ...)	Reads the value of an internal socket configuration option.
setsockopt(s, ...)	Changes the value of an internal socket configuration option.

The sockets API lets you create TCP endpoint data structures, connect these endpoints to remote server TCP endpoints, and read and write data streams. The TCP API hides all the details of the underlying network protocol handshaking and the segmentation and reassembly of the TCP data stream to and from IP packets.

In Figure 4-1, we showed how a web browser could download the *power-tools.html* web page from Joe's Hardware store using HTTP. The pseudocode in Figure 4-6 sketches how we might use the sockets API to highlight the steps the client and server could perform to implement this HTTP transaction.

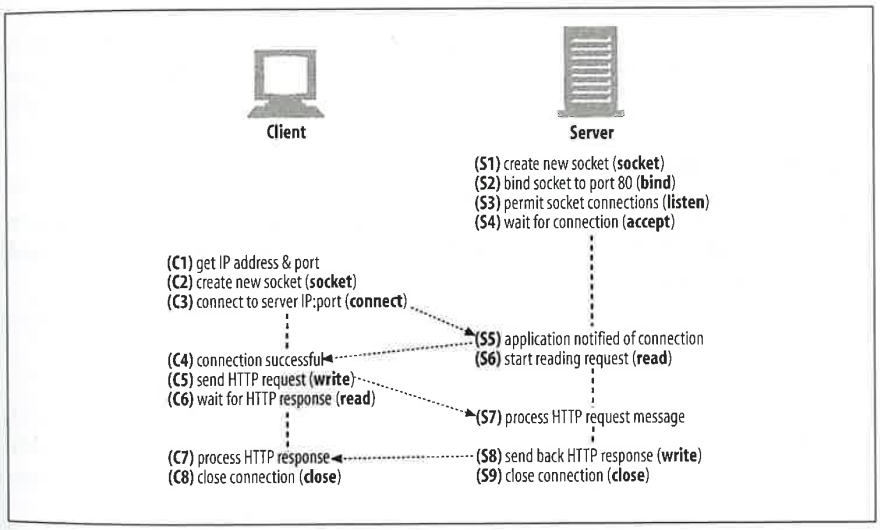


Figure 4-6. How TCP clients and servers communicate using the TCP sockets interface

We begin with the web server waiting for a connection (Figure 4-6, S4). The client determines the IP address and port number from the URL and proceeds to establish a TCP connection to the server (Figure 4-6, C3). Establishing a connection can take a while, depending on how far away the server is, the load on the server, and the congestion of the Internet.

Once the connection is set up, the client sends the HTTP request (Figure 4-6, C5) and the server reads it (Figure 4-6, S6). Once the server gets the entire request message, it processes the request, performs the requested action (Figure 4-6, S7), and writes the data back to the client. The client reads it (Figure 4-6, C6) and processes the response data (Figure 4-6, C7).

TCP Performance Considerations

Because HTTP is layered directly on TCP, the performance of HTTP transactions depends critically on the performance of the underlying TCP plumbing. This section highlights some significant performance considerations of these TCP connections. By understanding some of the basic performance characteristics of TCP, you'll better appreciate HTTP's connection optimization features, and you'll be able to design and implement higher-performance HTTP applications.

This section requires some understanding of the internal details of the TCP protocol. If you are not interested in (or are comfortable with) the details of TCP performance considerations, feel free to skip ahead to "HTTP Connection Handling." Because TCP is a complex topic, we can provide only a brief overview of TCP performance here. Refer to the section "For More Information" at the end of this chapter for a list of excellent TCP references.

HTTP Transaction Delays

Let's start our TCP performance tour by reviewing what networking delays occur in the course of an HTTP request. Figure 4-7 depicts the major connect, transfer, and processing delays for an HTTP transaction.

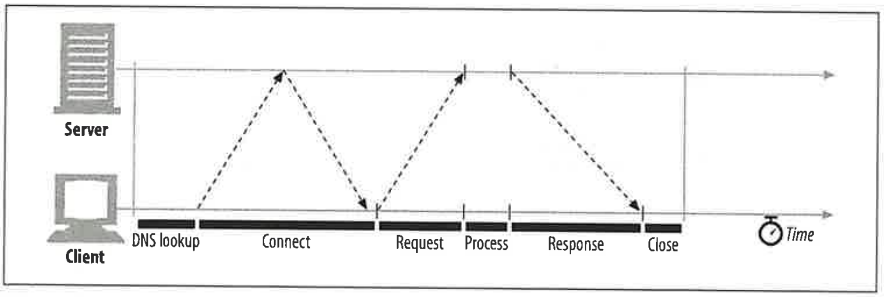


Figure 4-7. Timeline of a serial HTTP transaction

Notice that the transaction processing time can be quite small compared to the time required to set up TCP connections and transfer the request and response messages. Unless the client or server is overloaded or executing complex dynamic resources, most HTTP delays are caused by TCP network delays.

There are several possible causes of delay in an HTTP transaction:

1. A client first needs to determine the IP address and port number of the web server from the URI. If the hostname in the URI was not recently visited, it may take tens of seconds to convert the hostname from a URI into an IP address using the DNS resolution infrastructure.*
2. Next, the client sends a TCP connection request to the server and waits for the server to send back a connection acceptance reply. Connection setup delay occurs for every new TCP connection. This usually takes at most a second or two, but it can add up quickly when hundreds of HTTP transactions are made.
3. Once the connection is established, the client sends the HTTP request over the newly established TCP pipe. The web server reads the request message from the TCP connection as the data arrives and processes the request. It takes time for the request message to travel over the Internet and get processed by the server.
4. The web server then writes back the HTTP response, which also takes time.

The magnitude of these TCP network delays depends on hardware speed, the load of the network and server, the size of the request and response messages, and the distance between client and server. The delays also are significantly affected by technical intricacies of the TCP protocol.

Performance Focus Areas

The remainder of this section outlines some of the most common TCP-related delays affecting HTTP programmers, including the causes and performance impacts of:

- The TCP connection setup handshake
- TCP slow-start congestion control
- Nagle's algorithm for data aggregation
- TCP's delayed acknowledgment algorithm for piggybacked acknowledgments
- TIME_WAIT delays and port exhaustion

If you are writing high-performance HTTP software, you should understand each of these factors. If you don't need this level of performance optimization, feel free to skip ahead.

* Luckily, most HTTP clients keep a small DNS cache of IP addresses for recently accessed sites. When the IP address is already "cached" (recorded) locally, the lookup is instantaneous. Because most web browsing is to a small number of popular sites, hostnames usually are resolved very quickly.

TCP Connection Handshake Delays

When you set up a new TCP connection, even before you send any data, the TCP software exchanges a series of IP packets to negotiate the terms of the connection (see Figure 4-8). These exchanges can significantly degrade HTTP performance if the connections are used for small data transfers.

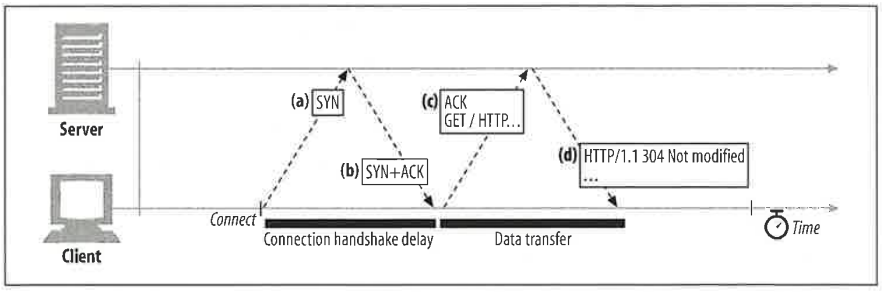


Figure 4-8. TCP requires two packet transfers to set up the connection before it can send data

Here are the steps in the TCP connection handshake:

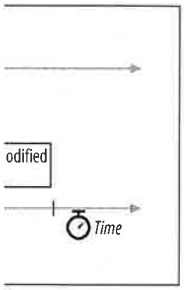
1. To request a new TCP connection, the client sends a small TCP packet (usually 40–60 bytes) to the server. The packet has a special "SYN" flag set, which means it's a connection request. This is shown in Figure 4-8a.
2. If the server accepts the connection, it computes some connection parameters and sends a TCP packet back to the client, with both the "SYN" and "ACK" flags set, indicating that the connection request is accepted (see Figure 4-8b).
3. Finally, the client sends an acknowledgment back to the server, letting it know that the connection was established successfully (see Figure 4-8c). Modern TCP stacks let the client send data in this acknowledgment packet.

The HTTP programmer never sees these packets—they are managed invisibly by the TCP/IP software. All the HTTP programmer sees is a delay when creating a new TCP connection.

The SYN/SYN+ACK handshake (Figure 4-8a and b) creates a measurable delay when HTTP transactions do not exchange much data, as is commonly the case. The TCP connect ACK packet (Figure 4-8c) often is large enough to carry the entire HTTP request message,* and many HTTP server response messages fit into a single IP packet (e.g., when the response is a small HTML file of a decorative graphic, or a 304 Not Modified response to a browser cache request).

* IP packets are usually a few hundred bytes for Internet traffic and around 1,500 bytes for local traffic.

ıy data, the TCP
ıf the connection
erformance if the



can send data

P packet (usually
set, which means

ction parameters
YN" and "ACK"
Figure 4-8b).
:, letting it know
lc). Modern TCP

d invisibly by the
ating a new TCP

neasurable delay
nly the case. The
carry the entire
s fit into a single
tive graphic, or a

for local traffic.

The end result is that small HTTP transactions may spend 50% or more of their time doing TCP setup. Later sections will discuss how HTTP allows reuse of existing connections to eliminate the impact of this TCP setup delay.

Delayed Acknowledgments

Because the Internet itself does not guarantee reliable packet delivery (Internet routers are free to destroy packets at will if they are overloaded), TCP implements its own acknowledgment scheme to guarantee successful data delivery.

Each TCP segment gets a sequence number and a data-integrity checksum. The receiver of each segment returns small acknowledgment packets back to the sender when segments have been received intact. If a sender does not receive an acknowledgment within a specified window of time, the sender concludes the packet was destroyed or corrupted and resends the data.

Because acknowledgments are small, TCP allows them to "piggyback" on outgoing data packets heading in the same direction. By combining returning acknowledgments with outgoing data packets, TCP can make more efficient use of the network. To increase the chances that an acknowledgment will find a data packet headed in the same direction, many TCP stacks implement a "delayed acknowledgment" algorithm. Delayed acknowledgments hold outgoing acknowledgments in a buffer for a certain window of time (usually 100–200 milliseconds), looking for an outgoing data packet on which to piggyback. If no outgoing data packet arrives in that time, the acknowledgment is sent in its own packet.

Unfortunately, the bimodal request-reply behavior of HTTP reduces the chances that piggybacking can occur. There just aren't many packets heading in the reverse direction when you want them. Frequently, the disabled acknowledgment algorithms introduce significant delays. Depending on your operating system, you may be able to adjust or disable the delayed acknowledgment algorithm.

Before you modify any parameters of your TCP stack, be sure you know what you are doing. Algorithms inside TCP were introduced to protect the Internet from poorly designed applications. If you modify any TCP configurations, be absolutely sure your application will not create the problems the algorithms were designed to avoid.

TCP Slow Start

The performance of TCP data transfer also depends on the *age* of the TCP connection. TCP connections "tune" themselves over time, initially limiting the maximum speed of the connection and increasing the speed over time as data is transmitted successfully. This tuning is called *TCP slow start*, and it is used to prevent sudden overloading and congestion of the Internet.

TCP slow start throttles the number of packets a TCP endpoint can have in flight at any one time. Put simply, each time a packet is received successfully, the sender gets permission to send two more packets. If an HTTP transaction has a large amount of data to send, it cannot send all the packets at once. It must send one packet and wait for an acknowledgment; then it can send two packets, each of which must be acknowledged, which allows four packets, etc. This is called "opening the congestion window."

Because of this congestion-control feature, new connections are slower than "tuned" connections that already have exchanged a modest amount of data. Because tuned connections are faster, HTTP includes facilities that let you reuse existing connections. We'll talk about these HTTP "persistent connections" later in this chapter.

Nagle's Algorithm and TCP_NODELAY

TCP has a data stream interface that permits applications to stream data of any size to the TCP stack—even a single byte at a time! But because each TCP segment carries at least 40 bytes of flags and headers, network performance can be degraded severely if TCP sends large numbers of packets containing small amounts of data.*

Nagle's algorithm (named for its creator, John Nagle) attempts to bundle up a large amount of TCP data before sending a packet, aiding network efficiency. The algorithm is described in RFC 896, "Congestion Control in IP/TCP Internetworks."

Nagle's algorithm discourages the sending of segments that are not full-size (a maximum-size packet is around 1,500 bytes on a LAN, or a few hundred bytes across the Internet). Nagle's algorithm lets you send a non-full-size packet only if all other packets have been acknowledged. If other packets are still in flight, the partial data is buffered. This buffered data is sent only when pending packets are acknowledged or when the buffer has accumulated enough data to send a full packet.†

Nagle's algorithm causes several HTTP performance problems. First, small HTTP messages may not fill a packet, so they may be delayed waiting for additional data that will never arrive. Second, Nagle's algorithm interacts poorly with disabled acknowledgments—Nagle's algorithm will hold up the sending of data until an acknowledgment arrives, but the acknowledgment itself will be delayed 100–200 milliseconds by the delayed acknowledgment algorithm.‡

HTTP applications often disable Nagle's algorithm to improve performance, by setting the TCP_NODELAY parameter on their stacks. If you do this, you must ensure that you write large chunks of data to TCP so you don't create a flurry of small packets.

* Sending a storm of single-byte packets is called "sender silly window syndrome." This is inefficient, antisocial, and can be disruptive to other Internet traffic.

† Several variations of this algorithm exist, including timeouts and acknowledgment logic changes, but the basic algorithm causes buffering of data smaller than a TCP segment.

‡ These problems can become worse when using pipelined connections (described later in this chapter), because clients may have several messages to send to the same server and do not want delays.

TIME_WAIT Accumulation and Port Exhaustion

TIME_WAIT port exhaustion is a serious performance problem that affects performance benchmarking but is relatively uncommon in real deployments. It warrants special attention because most people involved in performance benchmarking eventually run into this problem and get unexpectedly poor performance.

When a TCP endpoint closes a TCP connection, it maintains in memory a small control block recording the IP addresses and port numbers of the recently closed connection. This information is maintained for a short time, typically around twice the estimated maximum segment lifetime (called "2MSL"; often two minutes*), to make sure a new TCP connection with the same addresses and port numbers is not created during this time. This prevents any stray duplicate packets from the previous connection from accidentally being injected into a new connection that has the same addresses and port numbers. In practice, this algorithm prevents two connections with the exact same IP addresses and port numbers from being created, closed, and recreated within two minutes.

Today's higher-speed routers make it extremely unlikely that a duplicate packet will show up on a server's doorstep minutes after a connection closes. Some operating systems set 2MSL to a smaller value, but be careful about overriding this value. Packets do get duplicated, and TCP data will be corrupted if a duplicate packet from a past connection gets inserted into a new stream with the same connection values.

The 2MSL connection close delay normally is not a problem, but in benchmarking situations, it can be. It's common that only one or a few test load-generation computers are connecting to a system under benchmark test, which limits the number of client IP addresses that connect to the server. Furthermore, the server typically is listening on HTTP's default TCP port, 80. These circumstances limit the available combinations of connection values, at a time when port numbers are blocked from reuse by TIME_WAIT.

In a pathological situation with one client and one web server, of the four values that make up a TCP connection:

```
<source-IP-address, source-port, destination-IP-address, destination-port>
```

three of them are fixed—only the source port is free to change:

```
<client-IP, source-port, server-IP, 80>
```

Each time the client connects to the server, it gets a new source port in order to have a unique connection. But because a limited number of source ports are available (say, 60,000) and no connection can be reused for 2MSL seconds (say, 120 seconds), this limits the connect rate to 60,000 / 120 = 500 transactions/sec. If you keep

* The 2MSL value of two minutes is historical. Long ago, when routers were much slower, it was estimated that a duplicate copy of a packet might be able to remain queued in the Internet for up to a minute before being destroyed. Today, the maximum segment lifetime is much smaller.

making optimizations, and your server doesn't get faster than about 500 transactions/sec, make sure you are not experiencing TIME_WAIT port exhaustion. You can fix this problem by using more client load-generator machines or making sure the client and server rotate through several virtual IP addresses to add more connection combinations.

Even if you do not suffer port exhaustion problems, be careful about having large numbers of open connections or large numbers of control blocks allocated for connection in wait states. Some operating systems slow down dramatically when there are numerous open connections or control blocks.

HTTP Connection Handling

The first two sections of this chapter provided a fire-hose tour of TCP connections and their performance implications. If you'd like to learn more about TCP networking, check out the resources listed at the end of the chapter.

We're going to switch gears now and get squarely back to HTTP. The rest of this chapter explains the HTTP technology for manipulating and optimizing connections. We'll start with the HTTP Connection header, an often misunderstood but important part of HTTP connection management. Then we'll talk about HTTP's connection optimization techniques.

The Oft-Misunderstood Connection Header

HTTP allows a chain of HTTP intermediaries between the client and the ultimate origin server (proxies, caches, etc.). HTTP messages are forwarded hop by hop from the client, through intermediary devices, to the origin server (or the reverse).

In some cases, two adjacent HTTP applications may want to apply a set of options to their shared connection. The HTTP Connection header field has a comma-separated list of *connection tokens* that specify options for the connection that aren't propagated to other connections. For example, a connection that must be closed after sending the next message can be indicated by Connection: close.

The Connection header sometimes is confusing, because it can carry three different types of tokens:

- HTTP header field names, listing headers relevant for only this connection
- Arbitrary token values, describing nonstandard options for this connection
- The value `close`, indicating the persistent connection will be closed when done

If a connection token contains the name of an HTTP header field, that header field contains connection-specific information and must not be forwarded. Any header fields listed in the Connection header must be deleted before the message is forwarded. Placing a hop-by-hop header name in a Connection header is known as

"protecting the header," because the Connection header protects against accidental forwarding of the local header. An example is shown in Figure 4-9.

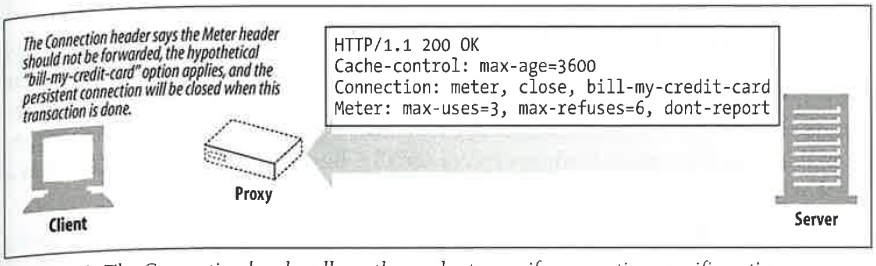


Figure 4-9. The Connection header allows the sender to specify connection-specific options

When an HTTP application receives a message with a Connection header, the receiver parses and applies all options requested by the sender. It then deletes the Connection header and all headers listed in the Connection header before forwarding the message to the next hop. In addition, there are a few hop-by-hop headers that might not be listed as values of a Connection header, but must not be proxied. These include Proxy-Authenticate, Proxy-Connection, Transfer-Encoding, and Upgrade. For more about the Connection header, see Appendix C.

Serial Transaction Delays

TCP performance delays can add up if the connections are managed naively. For example, suppose you have a web page with three embedded images. Your browser needs to issue four HTTP transactions to display this page: one for the top-level HTML and three for the embedded images. If each transaction requires a new connection, the connection and slow-start delays can add up (see Figure 4-10).*

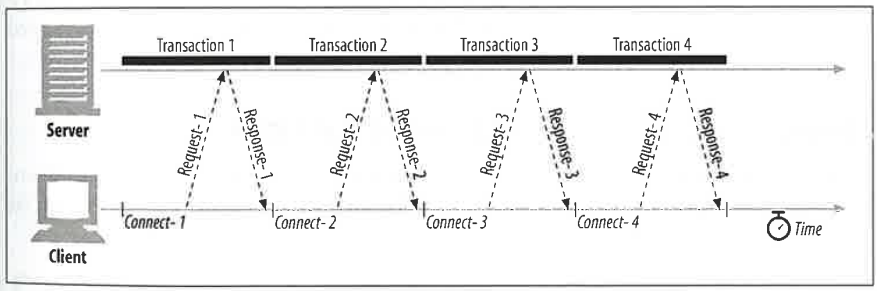


Figure 4-10. Four transactions (serial)

* For the purpose of this example, assume all objects are roughly the same size and are hosted from the same server, and that the DNS entry is cached, eliminating the DNS lookup time.

In addition to the real delay imposed by serial loading, there is also a psychological perception of slowness when a single image is loading and nothing is happening on the rest of the page. Users prefer multiple images to load at the same time.*

Another disadvantage of serial loading is that some browsers are unable to display anything onscreen until enough objects are loaded, because they don't know the sizes of the objects until they are loaded, and they may need the size information to decide where to position the objects on the screen. In this situation, the browser may be making good progress loading objects serially, but the user may be faced with a blank white screen, unaware that any progress is being made at all.†

Several current and emerging techniques are available to improve HTTP connection performance. The next several sections discuss four such techniques:

Parallel connections
: Concurrent HTTP requests across multiple TCP connections

Persistent connections
: Reusing TCP connections to eliminate connect/close delays

Pipelined connections
: Concurrent HTTP requests across a shared TCP connection

Multiplexed connections
: Interleaving chunks of requests and responses (experimental)

Parallel Connections

As we mentioned previously, a browser could naively process each embedded object serially by completely requesting the original HTML page, then the first embedded object, then the second embedded object, etc. But this is too slow!

HTTP allows clients to open multiple connections and perform multiple HTTP transactions in parallel, as sketched in Figure 4-11. In this example, four embedded images are loaded in parallel, with each transaction getting its own TCP connection.‡

Parallel Connections May Make Pages Load Faster

Composite pages consisting of embedded objects may load faster if they take advantage of the dead time and bandwidth limits of a single connection. The delays can be

* This is true even if loading multiple images at the same time is *slower* than loading images one at a time! Users often perceive multiple-image loading as faster.

† HTML designers can help eliminate this "layout delay" by explicitly adding width and height attributes to HTML tags for embedded objects such as images. Explicitly providing the width and height of the embedded image allows the browser to make graphical layout decisions before it receives the objects from the server.

‡ The embedded components do not all need to be hosted on the same web server, so the parallel connections can be established to multiple servers.

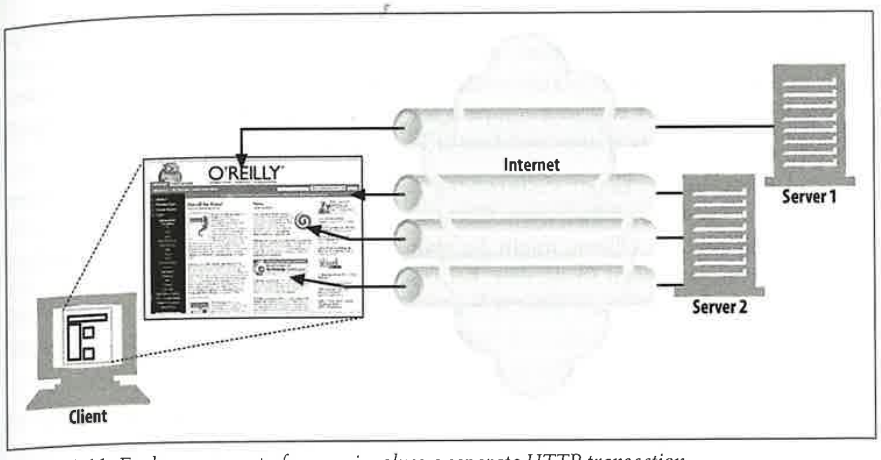


Figure 4-11. Each component of a page involves a separate HTTP transaction

overlapped, and if a single connection does not saturate the client's Internet bandwidth, the unused bandwidth can be allocated to loading additional objects.

Figure 4-12 shows a timeline for parallel connections, which is significantly faster than Figure 4-10. The enclosing HTML page is loaded first, and then the remaining three transactions are processed concurrently, each with their own connection.* Because the images are loaded in parallel, the connection delays are overlapped.

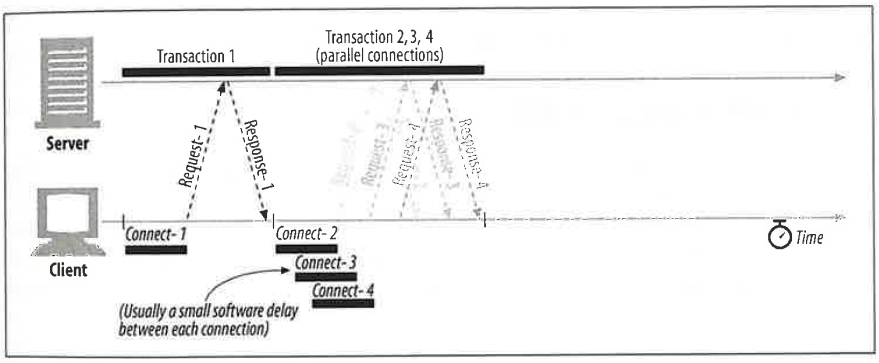


Figure 4-12. Four transactions (parallel)

Parallel Connections Are Not Always Faster

Even though parallel connections may be faster, however, they are not *always* faster. When the client's network bandwidth is scarce (for example, a browser connected to

* There will generally still be a small delay between each connection request due to software overheads, but the connection requests and transfer times are mostly overlapped.

the Internet through a 28.8-Kbps modem), most of the time might be spent just transferring data. In this situation, a single HTTP transaction to a fast server could easily consume all of the available modem bandwidth. If multiple objects are loaded in parallel, each object will just compete for this limited bandwidth, so each object will load proportionally slower, yielding little or no performance advantage.*

Also, a large number of open connections can consume a lot of memory and cause performance problems of their own. Complex web pages may have tens or hundreds of embedded objects. Clients might be able to open hundreds of connections, but few web servers will want to do that, because they often are processing requests for many other users at the same time. A hundred simultaneous users, each opening 100 connections, will put the burden of 10,000 connections on the server. This can cause significant server slowdown. The same situation is true for high-load proxies.

In practice, browsers do use parallel connections, but they limit the total number of parallel connections to a small number (often four). Servers are free to close excessive connections from a particular client.

Parallel Connections May "Feel" Faster

Okay, so parallel connections don't always make pages load faster. But even if they don't actually speed up the page transfer, as we said earlier, parallel connections often make users *feel* that the page loads faster, because they can see progress being made as multiple component objects appear onscreen in parallel.† Human beings perceive that web pages load faster if there's lots of action all over the screen, even if a stopwatch actually shows the aggregate page download time to be slower!

Persistent Connections

Web clients often open connections to the same site. For example, most of the embedded images in a web page often come from the same web site, and a significant number of hyperlinks to other objects often point to the same site. Thus, an application that initiates an HTTP request to a server likely will make more requests to that server in the near future (to fetch the inline images, for example). This property is called *site locality*.

For this reason, HTTP/1.1 (and enhanced versions of HTTP/1.0) allows HTTP devices to keep TCP connections open after transactions complete and to reuse the preexisting connections for future HTTP requests. TCP connections that are kept

* In fact, because of the extra overhead from multiple connections, it's quite possible that parallel connections could take longer to load the entire page than serial downloads.

† This effect is amplified by the increasing use of progressive images that produce low-resolution approximations of images first and gradually increase the resolution.

open after transactions complete are called *persistent* connections. Nonpersistent connections are closed after each transaction. Persistent connections stay open across transactions, until either the client or the server decides to close them.

By reusing an idle, persistent connection that is already open to the target server, you can avoid the slow connection setup. In addition, the already open connection can avoid the slow-start congestion adaptation phase, allowing faster data transfers.

Persistent Versus Parallel Connections

As we've seen, parallel connections can speed up the transfer of composite pages. But parallel connections have some disadvantages:

- Each transaction opens/closes a new connection, costing time and bandwidth.
- Each new connection has reduced performance because of TCP slow start.
- There is a practical limit on the number of open parallel connections.

Persistent connections offer some advantages over parallel connections. They reduce the delay and overhead of connection establishment, keep the connections in a tuned state, and reduce the potential number of open connections. However, persistent connections need to be managed with care, or you may end up accumulating a large number of idle connections, consuming local resources and resources on remote clients and servers.

Persistent connections can be most effective when used in conjunction with parallel connections. Today, many web applications open a small number of parallel connections, each persistent. There are two types of persistent connections: the older HTTP/1.0+ "keep-alive" connections and the modern HTTP/1.1 "persistent" connections. We'll look at both flavors in the next few sections.

HTTP/1.0+ Keep-Alive Connections

Many HTTP/1.0 browsers and servers were extended (starting around 1996) to support an early, experimental type of persistent connections called *keep-alive connections*. These early persistent connections suffered from some interoperability design problems that were rectified in later revisions of HTTP/1.1, but many clients and servers still use these earlier keep-alive connections.

Some of the performance advantages of keep-alive connections are visible in Figure 4-13, which compares the timeline for four HTTP transactions over serial connections against the same transactions over a single persistent connection. The timeline is compressed because the connect and close overheads are removed.*

* Additionally, the request and response time might also be reduced because of elimination of the slow-start phase. This performance benefit is not depicted in the figure.

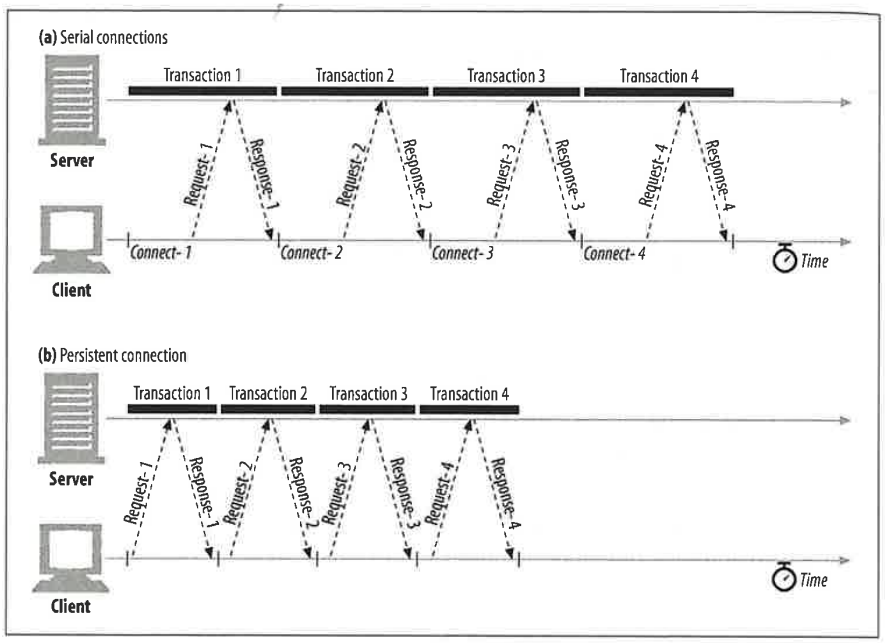


Figure 4-13. Four transactions (serial versus persistent)

Keep-Alive Operation

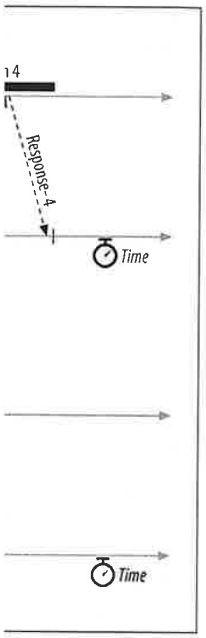
Keep-alive is deprecated and no longer documented in the current HTTP/1.1 specification. However, keep-alive handshaking is still in relatively common use by browsers and servers, so HTTP implementors should be prepared to interoperate with it. We'll take a quick look at keep-alive operation now. Refer to older versions of the HTTP/1.1 specification (such as RFC 2068) for a more complete explanation of keep-alive handshaking.

Clients implementing HTTP/1.0 keep-alive connections can request that a connection be kept open by including the Connection: Keep-Alive request header.

If the server is willing to keep the connection open for the next request, it will respond with the same header in the response (see Figure 4-14). If there is no Connection: keep-alive header in the response, the client assumes that the server does not support keep-alive and that the server will close the connection when the response message is sent back.

Keep-Alive Options

Note that the keep-alive headers are just requests to keep the connection alive. Clients and servers do not need to agree to a keep-alive session if it is requested. They



HTTP/1.1 specifi-
non use by brows-
teroperate with it.
der versions of the
ete explanation of

est that a connec-
header.

xt request, it will
f there is no Con-
at the server does
nection when the

nection alive. Cli-
is requested. They

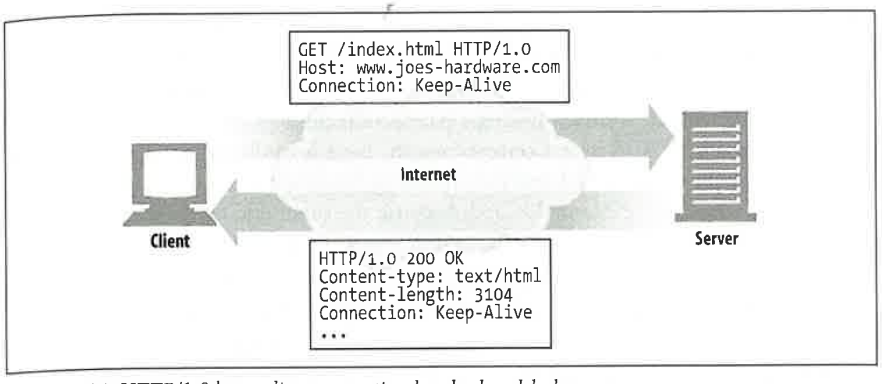


Figure 4-14. HTTP/1.0 keep-alive transaction header handshake

can close idle keep-alive connections at any time and are free to limit the number of transactions processed on a keep-alive connection.

The keep-alive behavior can be tuned by comma-separated options specified in the Keep-Alive general header:

- The `timeout` parameter is sent in a Keep-Alive response header. It estimates how long the server is likely to keep the connection alive for. This is not a guarantee.
- The `max` parameter is sent in a Keep-Alive response header. It estimates how many more HTTP transactions the server is likely to keep the connection alive for. This is not a guarantee.
- The Keep-Alive header also supports arbitrary unprocessed attributes, primarily for diagnostic and debugging purposes. The syntax is *name* [= *value*].

The Keep-Alive header is completely optional but is permitted only when Connection: Keep-Alive also is present. Here's an example of a Keep-Alive response header indicating that the server intends to keep the connection open for at most five more transactions, or until it has sat idle for two minutes:

```
Connection: Keep-Alive
Keep-Alive: max=5, timeout=120
```

Keep-Alive Connection Restrictions and Rules

Here are some restrictions and clarifications regarding the use of keep-alive connections:

- Keep-alive does not happen by default in HTTP/1.0. The client must send a Connection: Keep-Alive request header to activate keep-alive connections.
- The Connection: Keep-Alive header must be sent with all messages that want to continue the persistence. If the client does not send a Connection: Keep-Alive header, the server will close the connection after that request.

- Clients can tell if the server will close the connection after the response by detecting the absence of the Connection: Keep-Alive response header.
- The connection can be kept open only if the length of the message's entity body can be determined without sensing a connection close—this means that the entity body must have a correct Content-Length, have a multipart media type, or be encoded with the chunked transfer encoding. Sending the wrong Content-Length back on a keep-alive channel is bad, because the other end of the transaction will not be able to accurately detect the end of one message and the start of another.
- Proxies and gateways must enforce the rules of the Connection header; the proxy or gateway must remove any header fields named in the Connection header, and the Connection header itself, before forwarding or caching the message.
- Formally, keep-alive connections should not be established with a proxy server that isn't guaranteed to support the Connection header, to prevent the problem with dumb proxies described below. This is not always possible in practice.
- Technically, any Connection header fields (including Connection: Keep-Alive) received from an HTTP/1.0 device should be ignored, because they may have been forwarded mistakenly by an older proxy server. In practice, some clients and servers bend this rule, although they run the risk of hanging on older proxies.
- Clients must be prepared to retry requests if the connection closes before they receive the entire response, unless the request could have side effects if repeated.

Keep-Alive and Dumb Proxies

Let's take a closer look at the subtle problem with keep-alive and dumb proxies. A web client's Connection: Keep-Alive header is intended to affect just the single TCP link leaving the client. This is why it is named the "connection" header. If the client is talking to a web server, the client sends a Connection: Keep-Alive header to tell the server it wants keep-alive. The server sends a Connection: Keep-Alive header back if it supports keep-alive and doesn't send it if it doesn't.

The Connection header and blind relays

The problem comes with proxies—in particular, proxies that don't understand the Connection header and don't know that they need to remove the header before proxying it down the chain. Many older or simple proxies act as *blind relays*, tunneling bytes from one connection to another, without specially processing the Connection header.

Imagine a web client talking to a web server through a dumb proxy that is acting as a blind relay. This situation is depicted in Figure 4-15.

Here's what's going on in this figure:

1. In Figure 4-15a, a web client sends a message to the proxy, including the Connection: Keep-Alive header, requesting a keep-alive connection if possible. The client waits for a response to learn if its request for a keep-alive channel was granted.

ter the response by
e header.

nessage's entity body
means that the entity
t media type, or be
ong Content-Length
f the transaction will
ne start of another.

on header; the proxy
nection header, and
ne message.

with a proxy server
prevent the problem
ble in practice.

nection: Keep-Alive)
ause they may have
actice, some clients
ing on older proxies.

n closes before they
e effects if repeated.

nd dumb proxies. A
just the single TCP
header. If the client
ive header to tell the
Alive header back if

on't understand the
neader before proxy-
lays, tunneling bytes
Connection header.

xy that is acting as a

cluding the Connec-
possible. The client
nel was granted.

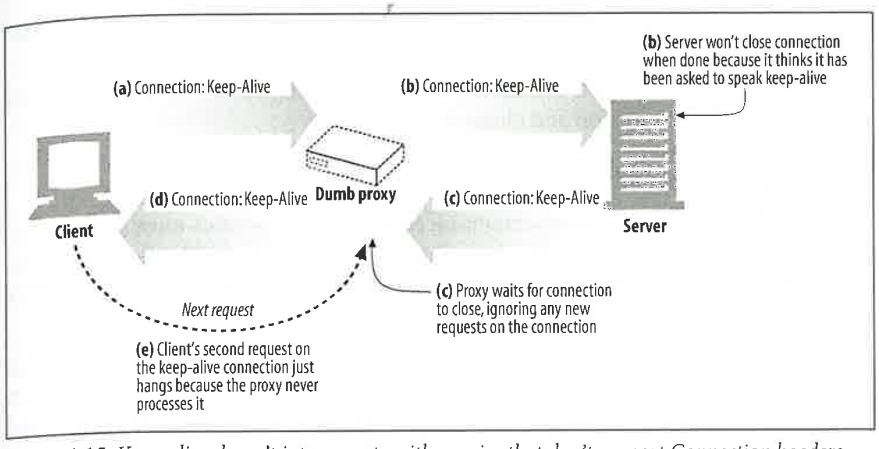


Figure 4-15. Keep-alive doesn't interoperate with proxies that don't support Connection headers

2. The dumb proxy gets the HTTP request, but it doesn't understand the Connection header (it just treats it as an extension header). The proxy has no idea what keep-alive is, so it passes the message verbatim down the chain to the server (Figure 4-15b). But the Connection header is a hop-by-hop header; it applies to only a single transport link and shouldn't be passed down the chain. Bad things are about to happen.
3. In Figure 4-15b, the relayed HTTP request arrives at the web server. When the web server receives the proxied Connection: Keep-Alive header, it mistakenly concludes that the proxy (which looks like any other client to the server) wants to speak keep-alive! That's fine with the web server—it agrees to speak keep-alive and sends a Connection: Keep-Alive response header back in Figure 4-15c. So, at this point, the web server thinks it is speaking keep-alive with the proxy and will adhere to rules of keep-alive. But the proxy doesn't know the first thing about keep-alive. Uh-oh.
4. In Figure 4-15d, the dumb proxy relays the web server's response message back to the client, passing along the Connection: Keep-Alive header from the web server. The client sees this header and assumes the proxy has agreed to speak keep-alive. So at this point, both the client and server believe they are speaking keep-alive, but the proxy they are talking to doesn't know anything about keep-alive.
5. Because the proxy doesn't know anything about keep-alive, it reflects all the data it receives back to the client and then waits for the origin server to close the connection. But the origin server will not close the connection, because it believes the proxy explicitly asked the server to keep the connection open. So the proxy will hang waiting for the connection to close.
6. When the client gets the response message back in Figure 4-15d, it moves right along to the next request, sending another request to the proxy on the keep-alive connection (see Figure 4-15e). Because the proxy never expects another request

on the same connection, the request is ignored and the browser just spins, making no progress.

7. This miscommunication causes the browser to hang until the client or server times out the connection and closes it.*

Proxies and hop-by-hop headers

To avoid this kind of proxy miscommunication, modern proxies must never proxy the Connection header or any headers whose names appear inside the Connection values. So if a proxy receives a Connection: Keep-Alive header, it shouldn't proxy either the Connection header or any headers named Keep-Alive.

In addition, there are a few hop-by-hop headers that might not be listed as values of a Connection header, but must not be proxied or served as a cache response either. These include Proxy-Authenticate, Proxy-Connection, Transfer-Encoding, and Upgrade. For more information, refer back to "The Oft-Misunderstood Connection Header."

The Proxy-Connection Hack

Browser and proxy implementors at Netscape proposed a clever workaround to the blind relay problem that didn't require all web applications to support advanced versions of HTTP. The workaround introduced a new header called Proxy-Connection and solved the problem of a single blind relay interposed directly after the client—but not all other situations. Proxy-Connection is implemented by modern browsers when proxies are explicitly configured and is understood by many proxies.

The idea is that dumb proxies get into trouble because they blindly forward hop-by-hop headers such as Connection: Keep-Alive. Hop-by-hop headers are relevant only for that single, particular connection and must not be forwarded. This causes trouble when the forwarded headers are misinterpreted by downstream servers as requests from the proxy itself to control its connection.

In the Netscape workaround, browsers send nonstandard Proxy-Connection extension headers to proxies, instead of officially supported and well-known Connection headers. If the proxy is a blind relay, it relays the nonsense Proxy-Connection header to the web server, which harmlessly ignores the header. But if the proxy is a smart proxy (capable of understanding persistent connection handshaking), it replaces the nonsense Proxy-Connection header with a Connection header, which is then sent to the server, having the desired effect.

Figure 4-16a–d shows how a blind relay harmlessly forwards Proxy-Connection headers to the web server, which ignores the header, causing no keep-alive connection to

* There are many similar scenarios where failures occur due to blind relays and forwarded handshaking.

be established between the client and proxy or the proxy and server. The smart proxy in Figure 4-16e–h understands the Proxy-Connection header as a request to speak keep-alive, and it sends out its own Connection: Keep-Alive headers to establish keep-alive connections.

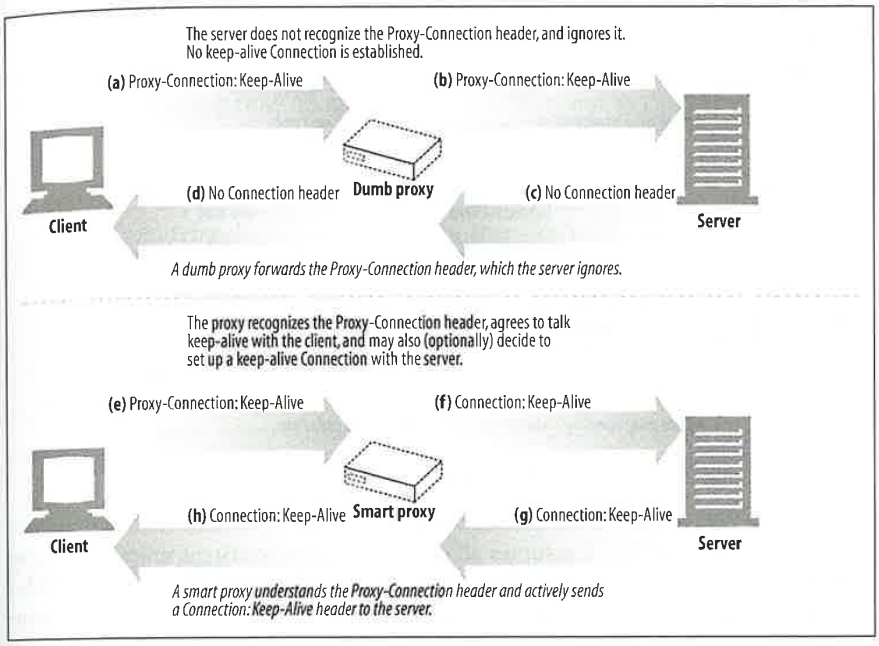


Figure 4-16. Proxy-Connection header fixes single blind relay

This scheme works around situations where there is only one proxy between the client and server. But if there is a smart proxy on either side of the dumb proxy, the problem will rear its ugly head again, as shown in Figure 4-17.

Furthermore, it is becoming quite common for "invisible" proxies to appear in networks, either as firewalls, intercepting caches, or reverse proxy server accelerators. Because these devices are invisible to the browser, the browser will not send them Proxy-Connection headers. It is critical that transparent web applications implement persistent connections correctly.

HTTP/1.1 Persistent Connections

HTTP/1.1 phased out support for keep-alive connections, replacing them with an improved design called *persistent connections*. The goals of persistent connections are the same as those of keep-alive connections, but the mechanisms behave better.

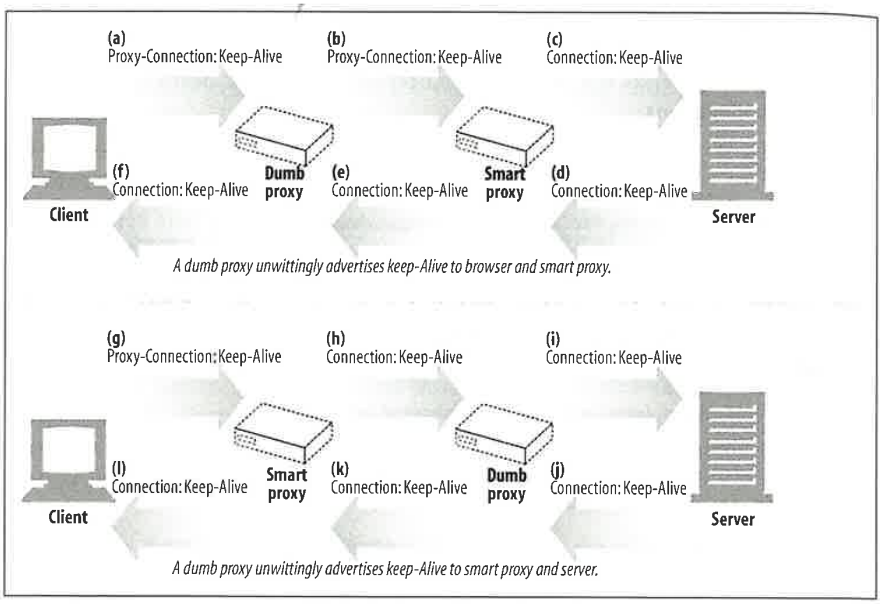


Figure 4-17. Proxy-Connection still fails for deeper hierarchies of proxies

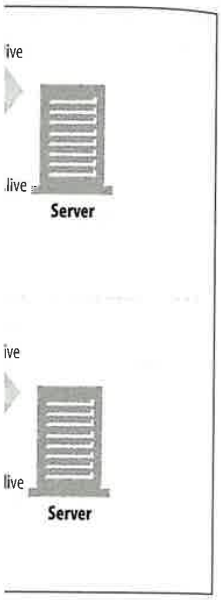
Unlike HTTP/1.0+ keep-alive connections, HTTP/1.1 persistent connections are active by default. HTTP/1.1 assumes *all* connections are persistent unless otherwise indicated. HTTP/1.1 applications have to explicitly add a Connection: close header to a message to indicate that a connection should close after the transaction is complete. This is a significant difference from previous versions of the HTTP protocol, where keep-alive connections were either optional or completely unsupported.

An HTTP/1.1 client assumes an HTTP/1.1 connection will remain open after a response, unless the response contains a Connection: close header. However, clients and servers still can close idle connections at any time. Not sending Connection: close does not mean that the server promises to keep the connection open forever.

Persistent Connection Restrictions and Rules

Here are the restrictions and clarifications regarding the use of persistent connections:

- After sending a Connection: close request header, the client can't send more requests on that connection.
- If a client does not want to send another request on the connection, it should send a Connection: close request header in the final request.
- The connection can be kept persistent only if all messages on the connection have a correct, self-defined message length—i.e., the entity bodies must have correct Content-Lengths or be encoded with the chunked transfer encoding.



ent connections are
ent unless otherwise
ection: close header
transaction is com-
the HTTP protocol,
unsupported.

emain open after a
er. However, clients
ending Connection:
ion open forever.

sistent connections:
nt can't send more

nnection, it should

on the connection
bodies must have
nsfer encoding.

- HTTP/1.1 proxies must manage persistent connections separately with clients and servers—each persistent connection applies to a single transport hop.
- HTTP/1.1 proxy servers should not establish persistent connections with an HTTP/1.0 client (because of the problems of older proxies forwarding Connection headers) unless they know something about the capabilities of the client. This is, in practice, difficult, and many vendors bend this rule.
- Regardless of the values of Connection headers, HTTP/1.1 devices may close the connection at any time, though servers should try not to close in the middle of transmitting a message and should always respond to at least one request before closing.
- HTTP/1.1 applications must be able to recover from asynchronous closes. Clients should retry the requests as long as they don't have side effects that could accumulate.
- Clients must be prepared to retry requests if the connection closes before they receive the entire response, unless the request could have side effects if repeated.
- A single user client should maintain at most two persistent connections to any server or proxy, to prevent the server from being overloaded. Because proxies may need more connections to a server to support concurrent users, a proxy should maintain at most $2N$ connections to any server or parent proxy, if there are N users trying to access the servers.

Pipelined Connections

HTTP/1.1 permits optional *request pipelining* over persistent connections. This is a further performance optimization over keep-alive connections. Multiple requests can be enqueued before the responses arrive. While the first request is streaming across the network to a server on the other side of the globe, the second and third requests can get underway. This can improve performance in high-latency network conditions, by reducing network round trips.

Figure 4-18a-c shows how persistent connections can eliminate TCP connection delays and how pipelined requests (Figure 4-18c) can eliminate transfer latencies.

There are several restrictions for pipelining:

- HTTP clients should not pipeline until they are sure the connection is persistent.
- HTTP responses must be returned in the same order as the requests. HTTP messages are not tagged with sequence numbers, so there is no way to match responses with requests if the responses are received out of order.
- HTTP clients must be prepared for the connection to close at any time and be prepared to redo any pipelined requests that did not finish. If the client opens a persistent connection and immediately issues 10 requests, the server is free to close the connection after processing only, say, 5 requests. The remaining 5

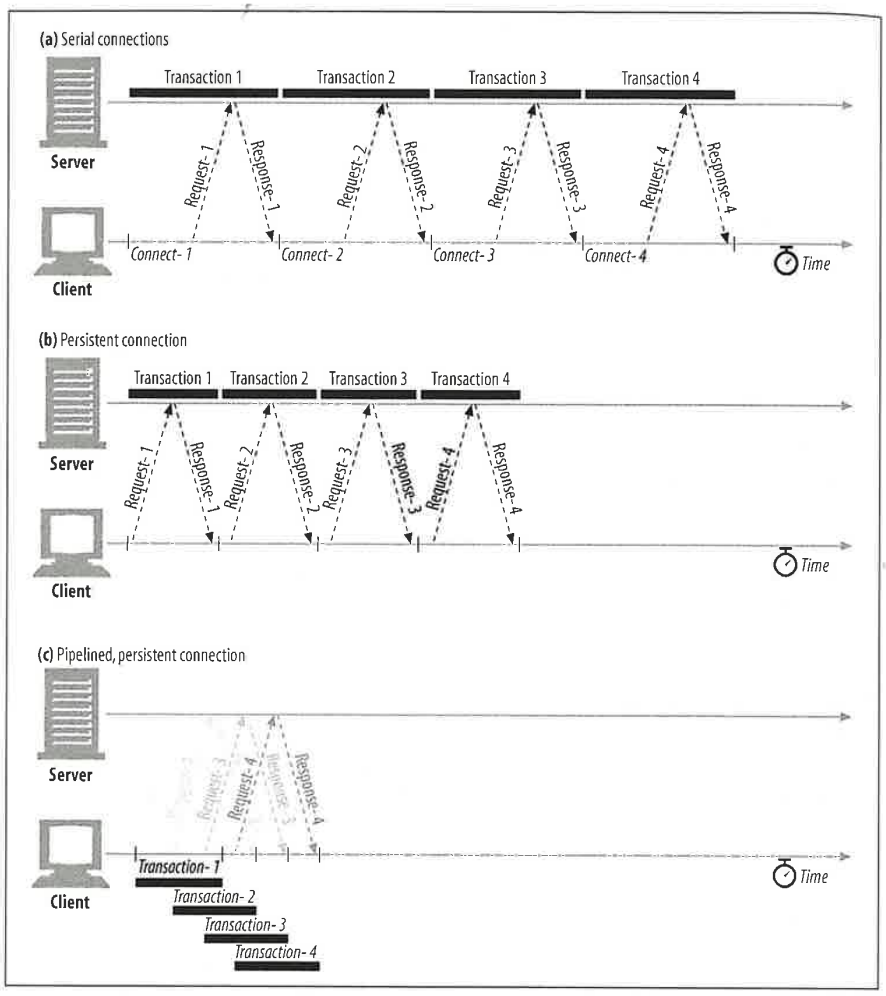
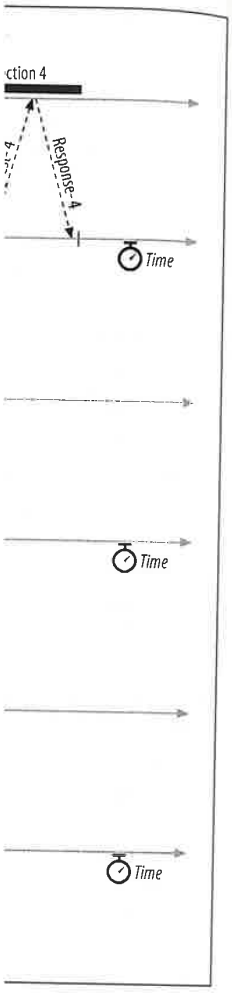


Figure 4-18. Four transactions (pipelined connections)

requests will fail, and the client must be willing to handle these premature closes and reissue the requests.

- HTTP clients should not pipeline requests that have side effects (such as POSTs). In general, on error, pipelining prevents clients from knowing which of a series of pipelined requests were executed by the server. Because nonidempotent requests such as POSTs cannot safely be retried, you run the risk of some methods never being executed in error conditions.



se premature closes

le effects (such as
n knowing which of
ecause nonidempo-
un the risk of some

The Mysteries of Connection Close

Connection management—particularly knowing when and how to close connections—is one of the practical black arts of HTTP. This issue is more subtle than many developers first realize, and little has been written on the subject.

"At Will" Disconnection

Any HTTP client, server, or proxy can close a TCP transport connection at any time. The connections normally are closed at the end of a message,* but during error conditions, the connection may be closed in the middle of a header line or in other strange places.

This situation is common with pipelined persistent connections. HTTP applications are free to close persistent connections after any period of time. For example, after a persistent connection has been idle for a while, a server may decide to shut it down.

However, the server can never know for sure that the client on the other end of the line wasn't about to send data at the same time that the "idle" connection was being shut down by the server. If this happens, the client sees a connection error in the middle of writing its request message.

Content-Length and Truncation

Each HTTP response should have an accurate Content-Length header to describe the size of the response body. Some older HTTP servers omit the Content-Length header or include an erroneous length, depending on a server connection close to signify the actual end of data.

When a client or proxy receives an HTTP response terminating in connection close, and the actual transferred entity length doesn't match the Content-Length (or there is no Content-Length), the receiver should question the correctness of the length.

If the receiver is a caching proxy, the receiver should not cache the response (to minimize future compounding of a potential error). The proxy should forward the questionable message intact, without attempting to "correct" the Content-Length, to maintain semantic transparency.

Connection Close Tolerance, Retries, and Idempotency

Connections can close at any time, even in non-error conditions. HTTP applications have to be ready to properly handle unexpected closes. If a transport connection closes while the client is performing a transaction, the client should reopen the

* Servers shouldn't close a connection in the middle of a response unless client or network failure is suspected.

connection and retry one time, unless the transaction has side effects. The situation is worse for pipelined connections. The client can enqueue a large number of requests, but the origin server can close the connection, leaving numerous requests unprocessed and in need of rescheduling.

Side effects are important. When a connection closes after some request data was sent but before the response is returned, the client cannot be 100% sure how much of the transaction actually was invoked by the server. Some transactions, such as GETting a static HTML page, can be repeated again and again without changing anything. Other transactions, such as POSTing an order to an online book store, shouldn't be repeated, or you may risk multiple orders.

A transaction is *idempotent* if it yields the same result regardless of whether it is executed once or many times. Implementors can assume the GET, HEAD, PUT, DELETE, TRACE, and OPTIONS methods share this property.* Clients shouldn't pipeline nonidempotent requests (such as POSTs). Otherwise, a premature termination of the transport connection could lead to indeterminate results. If you want to send a nonidempotent request, you should wait for the response status for the previous request.

Nonidempotent methods or sequences must not be retried automatically, although user agents may offer a human operator the choice of retrying the request. For example, most browsers will offer a dialog box when reloading a cached POST response, asking if you want to post the transaction again.

Graceful Connection Close

TCP connections are bidirectional, as shown in Figure 4-19. Each side of a TCP connection has an input queue and an output queue, for data being read or written. Data placed in the output of one side will eventually show up on the input of the other side.

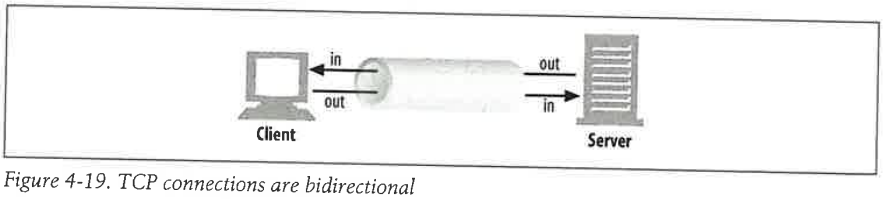


Figure 4-19. TCP connections are bidirectional

Full and half closes

An application can close either or both of the TCP input and output channels. A `close()` sockets call closes both the input and output channels of a TCP connection.

*Administrators who use GET-based dynamic forms should make sure the forms are idempotent.

This is called a "full close" and is depicted in Figure 4-20a. You can use the `shutdown()` sockets call to close either the input or output channel individually. This is called a "half close" and is depicted in Figure 4-20b.

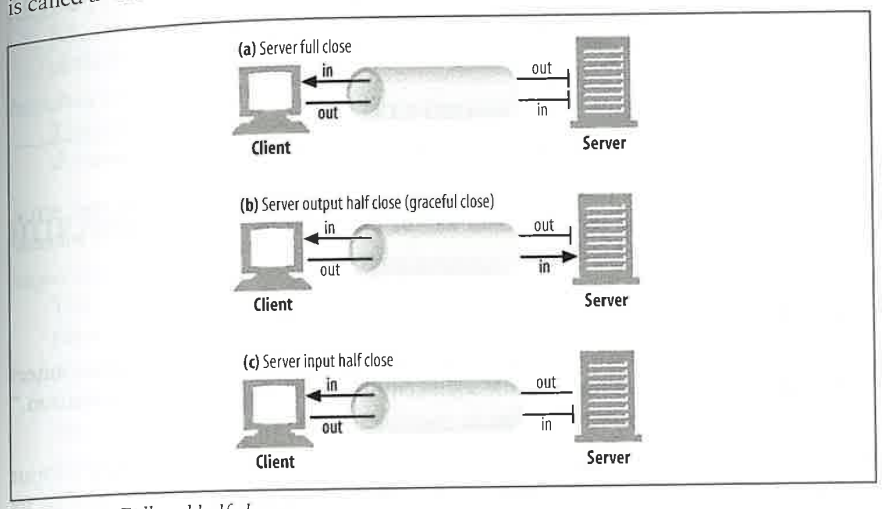


Figure 4-20. Full and half close

TCP close and reset errors

Simple HTTP applications can use only full closes. But when applications start talking to many other types of HTTP clients, servers, and proxies, and when they start using pipelined persistent connections, it becomes important for them to use half closes to prevent peers from getting unexpected write errors.

In general, closing the output channel of your connection is always safe. The peer on the other side of the connection will be notified that you closed the connection by getting an end-of-stream notification once all the data has been read from its buffer.

Closing the input channel of your connection is riskier, unless you know the other side doesn't plan to send any more data. If the other side sends data to your closed input channel, the operating system will issue a TCP "connection reset by peer" message back to the other side's machine, as shown in Figure 4-21. Most operating systems treat this as a serious error and erase any buffered data the other side has not read yet. This is very bad for pipelined connections.

Say you have sent 10 pipelined requests on a persistent connection, and the responses already have arrived and are sitting in your operating system's buffer (but the application hasn't read them yet). Now say you send request #11, but the server decides you've used this connection long enough, and closes it. Your request #11 will arrive at a closed connection and will reflect a reset back to you. This reset will erase your input buffers.

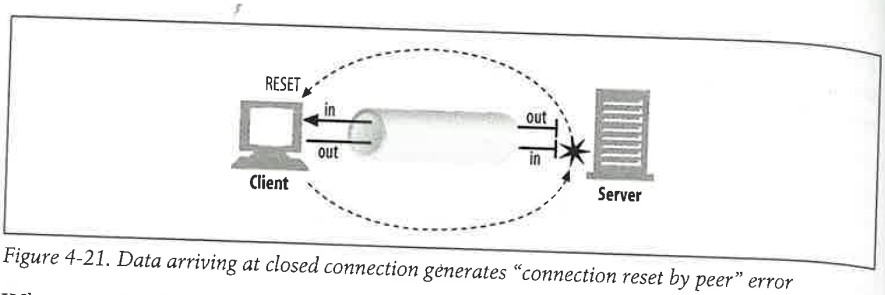


Figure 4-21. Data arriving at closed connection generates "connection reset by peer" error

When you finally get to reading data, you will get a connection reset by peer error, and the buffered, unread response data will be lost, even though much of it successfully arrived at your machine.

Graceful close

The HTTP specification counsels that when clients or servers want to close a connection unexpectedly, they should "issue a graceful close on the transport connection," but it doesn't describe how to do that.

In general, applications implementing graceful closes will first close their output channels and then wait for the peer on the other side of the connection to close *its* output channels. When both sides are done telling each other they won't be sending any more data (i.e., closing output channels), the connection can be closed fully, with no risk of reset.

Unfortunately, there is no guarantee that the peer implements or checks for half closes. For this reason, applications wanting to close gracefully should half close their output channels and periodically check the status of their input channels (looking for data or for the end of the stream). If the input channel isn't closed by the peer within some timeout period, the application may force connection close to save resources.

For More Information

This completes our overview of the HTTP plumbing trade. Please refer to the following reference sources for more information about TCP performance and HTTP connection-management facilities.

HTTP Connections

http://www.ietf.org/rfc/rfc2616.txt

RFC 2616, "Hypertext Transfer Protocol—HTTP/1.1," is the official specification for HTTP/1.1; it explains the usage of and HTTP header fields for implementing

parallel, persistent, and pipelined HTTP connections. This document does not cover the proper use of the underlying TCP connections.

http://www.ietf.org/rfc/rfc2068.txt
RFC 2068 is the 1997 version of the HTTP/1.1 protocol. It contains explanation of the HTTP/1.0+ Keep-Alive connections that is missing from RFC 2616.

http://www.ics.uci.edu/pub/ietf/http/draft-ietf-http-connection-00.txt
This expired Internet draft, "HTTP Connection Management," has some good discussion of issues facing HTTP connection management.

HTTP Performance Issues

http://www.w3.org/Protocols/HTTP/Performance/
This W3C web page, entitled "HTTP Performance Overview," contains a few papers and tools related to HTTP performance and connection management.

http://www.w3.org/Protocols/HTTP/1.0/HTTPPerformance.html
This short memo by Simon Spero, "Analysis of HTTP Performance Problems," is one of the earliest (1994) assessments of HTTP connection performance. The memo gives some early performance measurements of the effect of connection setup, slow start, and lack of connection sharing.

ftp://gatekeeper.dec.com/pub/DEC/WRL/research-reports/WRL-TR-95.4.pdf
"The Case for Persistent-Connection HTTP."

http://www.isi.edu/lsam/publications/phttp_tcp_interactions/paper.html
"Performance Interactions Between P-HTTP and TCP Implementations."

http://www.sun.com/sun-on-net/performance/tcp.slowstart.html
"TCP Slow Start Tuning for Solaris" is a web page from Sun Microsystems that talks about some of the practical implications of TCP slow start. It's a useful read, even if you are working with different operating systems.

TCP/IP

The following three books by W. Richard Stevens are excellent, detailed engineering texts on TCP/IP. These are extremely useful for anyone using TCP:

TCP Illustrated, Volume I: The Protocols
W. Richard Stevens, Addison Wesley

UNIX Network Programming, Volume 1: Networking APIs
W. Richard Stevens, Prentice-Hall

UNIX Network Programming, Volume 2: The Implementation
W. Richard Stevens, Prentice-Hall

The following papers and specifications describe TCP/IP and features that affect its performance. Some of these specifications are over 20 years old and, given the worldwide success of TCP/IP, probably can be classified as historical treasures:

http://www.acm.org/sigcomm/ccr/archive/2001/jan01/ccr-200101-mogul.pdf
: In "Rethinking the TCP Nagle Algorithm," Jeff Mogul and Greg Minshall present a modern perspective on Nagle's algorithm, outline what applications should and should not use the algorithm, and propose several modifications.

http://www.ietf.org/rfc/rfc2001.txt
: RFC 2001, "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms," defines the TCP slow-start algorithm.

http://www.ietf.org/rfc/rfc1122.txt
: RFC 1122, "Requirements for Internet Hosts—Communication Layers," discusses TCP acknowledgment and delayed acknowledgments.

http://www.ietf.org/rfc/rfc896.txt
: RFC 896, "Congestion Control in IP/TCP Internetworks," was released by John Nagle in 1984. It describes the need for TCP congestion control and introduces what is now called "Nagle's algorithm."

http://www.ietf.org/rfc/rfc0813.txt
: RFC 813, "Window and Acknowledgement Strategy in TCP," is a historical (1982) specification that describes TCP window and acknowledgment implementation strategies and provides an early description of the delayed acknowledgment technique.

http://www.ietf.org/rfc/rfc0793.txt
: RFC 793, "Transmission Control Protocol," is Jon Postel's classic 1981 definition of the TCP protocol.

features that affect its
. and, given the world-
treasures:

l-mogul.pdf
l and Greg Minshall
ine what applications
ral modifications.

Retransmit, and Fast
1.

ication Layers," dis-
s.

was released by John
ntrol and introduces

'CP," is a historical
nowledgment imple-
ne delayed acknowl-

classic 1981 defini-

PART II
HTTP Architecture

The six chapters of Part II highlight the HTTP server, proxy, cache, gateway, and robot applications, which are the building blocks of web systems architecture:

- Chapter 5, *Web Servers*, gives an overview of web server architectures.
- Chapter 6, *Proxies*, describes HTTP proxy servers, which are intermediary servers that connect HTTP clients and act as platforms for HTTP services and controls.
- Chapter 7, *Caching*, delves into the science of web caches—devices that improve performance and reduce traffic by making local copies of popular documents.
- Chapter 8, *Integration Points: Gateways, Tunnels, and Relays*, explains applications that allow HTTP to interoperate with software that speaks different protocols, including SSL encrypted protocols.
- Chapter 9, *Web Robots*, wraps up our tour of HTTP architecture with web clients.
- Chapter 10, *HTTP-NG*, covers future topics for HTTP—in particular, HTTP-NG.

Web Programming

O'REILLY®

HTTP: The Definitive Guide

Behind every successful web transaction lurks the Hypertext Transfer Protocol (HTTP), the language by which web clients and servers exchange documents and information. HTTP is commonly known as the workhorse behind the browsers we use every day to access our company intranets, locate out-of-print books, or research census information. But HTTP is used for far more than browsing the Web: the simplicity and ubiquity of HTTP also have made it the choice protocol for many other networked applications, most notably through web services such as SOAP and XML-RPC.

As the title suggests, *HTTP: The Definitive Guide* explains the HTTP protocol: how it works and how to use it to develop web-based applications. However, this book is not just about HTTP; it's also about all the other core Internet technologies that HTTP depends on to work effectively. Although HTTP is at the center of the book, the essence of *HTTP: The Definitive Guide* is in understanding how the Web works and how to apply that knowledge to web programming and administration. The book explains the technical workings, motivations, performance considerations, and objectives of HTTP and the technologies around which it revolves.

HTTP: The Definitive Guide is the bible for the HTTP protocol and related web technologies. Topics covered include:

- HTTP methods, headers, and status codes
- Optimizing proxies and caches
- Strategies for designing web robots and crawlers
- Cookies, authentication, and Secure HTTP
- Internationalization and content negotiation
- Redirection and load-balancing strategies

Written by experts with years of practical experience, this book uses clear, concise language and a plethora of detailed illustrations to help readers visualize what goes on behind the scenes, providing a complete understanding of the story behind each query on the Web.

All web programmers, administrators, and application developers need to be familiar with HTTP are many books that explain how to use the Web, but this is the vorks.