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 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

- 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.[16]

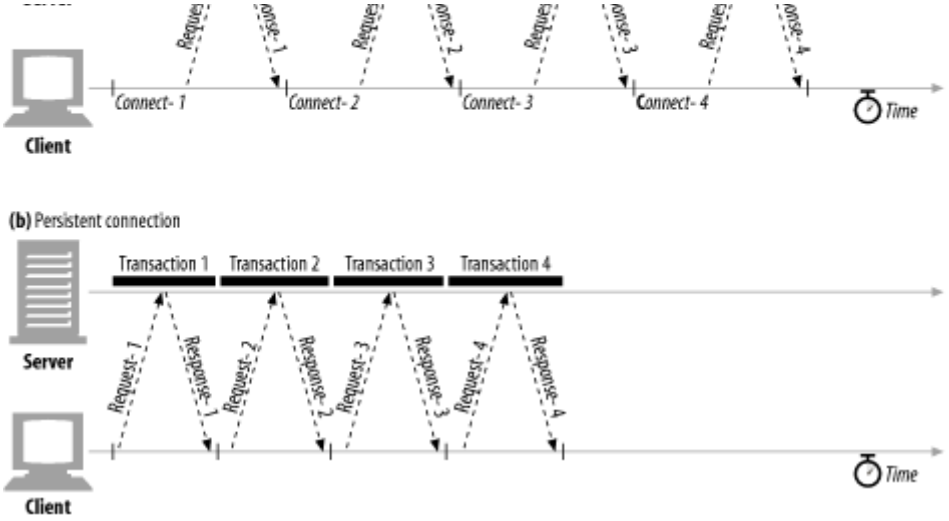

Figure 4-13. Four transactions (serial versus persistent)

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.

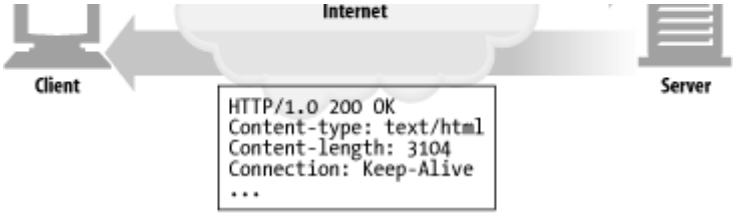

Figure 4-14. HTTP/1.0 keep-alive transaction header handshake

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 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:

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 chunkedtransfer 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.

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.

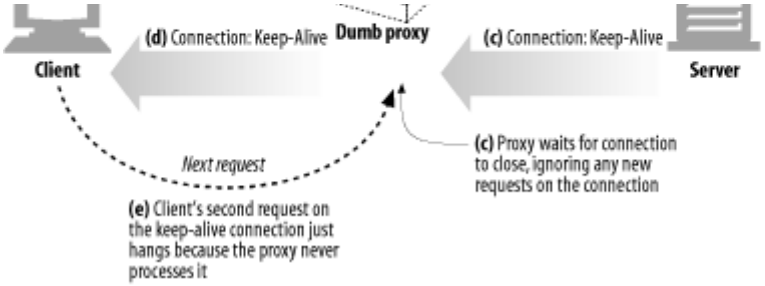

Figure 4-15. Keep-alive doesn't interoperate with proxies that don't support Connection headers

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.

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

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.[17]

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 Section 4.3.1.

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-

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 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.

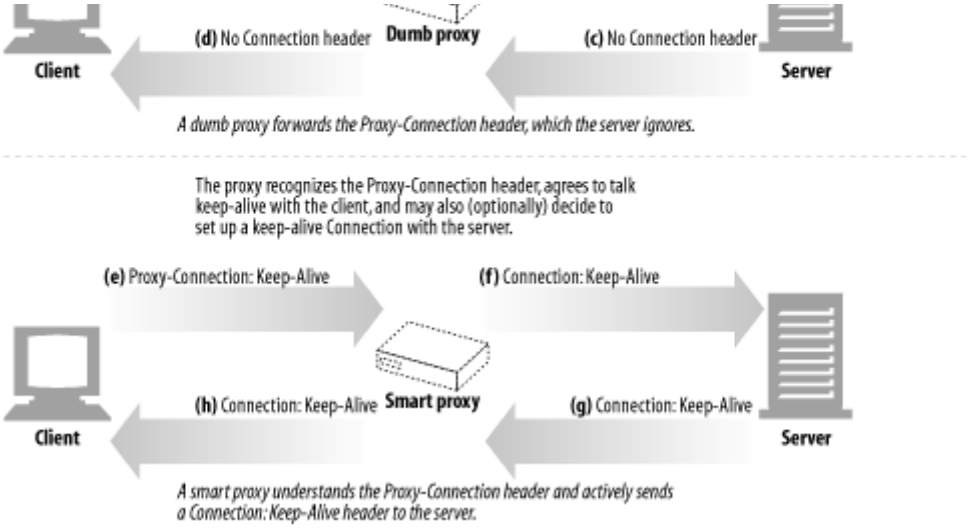

Figure 4-16. Proxy-Connection header fixes single blind relay

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.

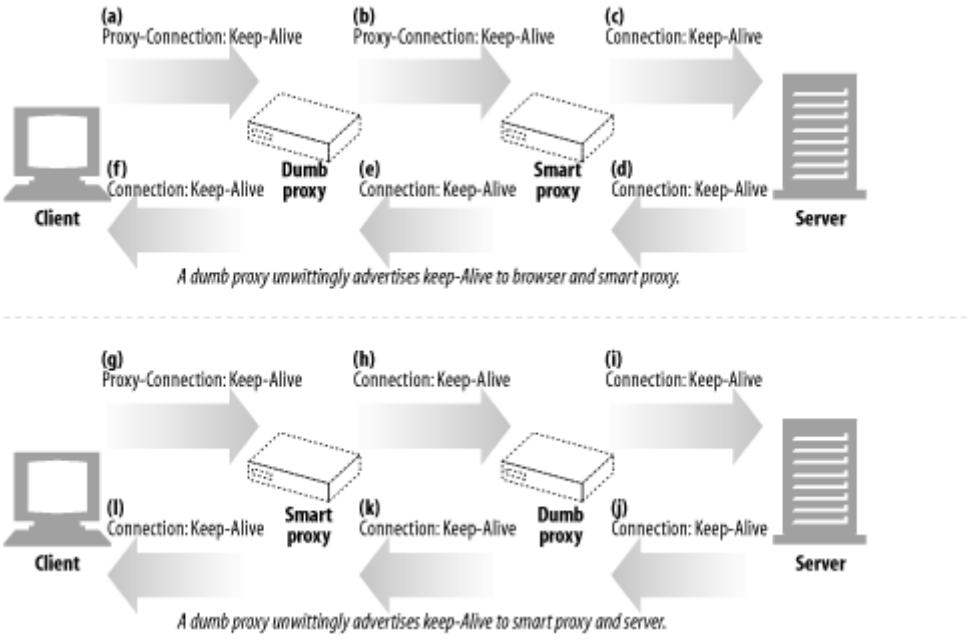

Figure 4-17. Proxy-Connection still fails for deeper hierarchies of proxies

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

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.

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

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 2*N* connections to any server or parent proxy, if there are *N* users trying to access the servers.

[16] 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.

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