Analysis of HTTP Performance problems

Simon E Spero

Abstract

This paper is the first in a series on performance issues in the World Wide Web.

HTTP is a transfer protocol used by the World Wide Web distributed hypermedia system to retrieve distributed objects. HTTP uses TCP as a transport layer. Certain design features of HTTP interact badly with TCP, causing problems with performance and with server scalability. Latency problems are caused by opening a single connection per request, through connection setup and slow-start costs. Further avoidable latency is incurred due to the protocol only returning a single object per request. Scalability problems are caused by TCP requring a server to maintain state for all recently closed connections.

Summary of HTTP/1.0

HTTP [http] is a transfer protocol used by the World Wide Web to retrieve information from distributed servers. The HTTP model is extremely simple; the client establishes a connection to the remote server, then issues a request. The server then processes the request, returns a response, and closes the connection.

Requests

The request format for HTTP is quite simple. The first line specifies an object, together with the name of an object to apply the method to. The most commonly used method is "GET", which ask the server to send a copy of the object to the client.

The client can also send a series of optional headers; these headers are in RFC-822 format. The most common headers are "Accept", which tells the server which object types the client can handle, and "User-Agent", which gives the implementation name of the client.

Request syntax

```
<METHOD> <URI> "HTTP/1.0" <crlf>
{<Header>: <Value> <crlf>}*
<crlf>
```

Example

```
GET  /index.html HTTP/1.0
Accept: text/plain
Accept: text/html
Accept: */*
User-Agent:  Yet Another User Agent
```

Responses

The response format is also quite simple. Responses start with a status line indicating which version of HTTP the server is running, together with a result code and an optional message.

This is followed by a series of optional object headers; the most important of these are "Content-Type", which describes the type of the object being returned, and "Content-Length", which indicates the length.The headers are teminated by an empty line.

The server now sends any requested data. After the data have been sent, the server drops the connection.

Response syntax

```
"HTTP/1.0" <result-code> [<message>]  <crlf>
{<Header>: <Value> <crlf>}*
<crlf>
```

Example

```
HTTP/1.0 200 OK
Server: MDMA/0.1
MIME-version: 1.0
Content-type: text/html
Last-Modified: Thu Jul  7 00:25:33 1994
Content-Length: 2003

<title>MDMA - Multithreaded Daemon for Multimedia Access</title>
<hr>
....
<hr>
<h2> MDMA - The speed daemon </h2>
<hr>

[Connection closed by foreign host]
```

Access Patterns

HTTP accesses usually exhibit a common pattern of behaviour. A client requests a hypertext page, then issues a sequence of requests to retrieve any icons referenced in the first document. Once the client has retrieved the icons, the user will typically select a hypertext link to follow. In the majority of cases, the referenced page will be on the same server as the original page.

HTTP Illustrated

The problems with HTTP can best be understood by looking at the network traffic generated by a typical HTTP transaction. This example was generated by using Van Jacobsen's tcpdump program to monitor a client at UNC fetching a copy of the NCSA Home page. This page is 1668 bytes long, including response headers. The client at UNC was a Sun Sparcstation 20/512, running Solaris 2.3. The server at NCSA was a Hewlett Packard 9000/735 running HP-UX.

The headers used in the request shown were captured from an xmosaic request. The headers consisted of 42 lines totaling around 1130 bytes; of these lines, 41 were "Accept".

Stage 1: Time = 0

The trace begins with the client sending a connection request to the http port on the server

```
.00000 unc.32840 > ncsa.80: S 2785173504:2785173504(0) win 8760 <mss 1460> (DF)
```

Stage 2: Time = 0.077

The server responds to the connect request with a connect response. The client acknowledges the connect

response, and send the first 536 bytes of the request.

```
.07769 ncsa.80 > unc.32840: S 530752000:530752000(0) ack 2785173505 win 8192
.07784 unc.32840 > ncsa.80: . ack 1 win 9112 (DF)
.07989 unc.32840 > ncsa.80: P 1:537(536) ack 1 win 9112 (DF)
```

Stage 3: Time = 0.35

The server acknowledges the first part of the request. The client then sends the second part, and without waiting for a response, follows up with the third and final part of the request.

```
.35079 ncsa.80 > unc.32840: . ack 537 win 8192
.35092 unc.32840 > ncsa.80: . 537:1073(536) ack 1 win 9112 (DF)
.35104 unc.32840 > ncsa.80: P 1073:1147(74) ack 1 win 9112 (DF)
```

Stage 4: Time = 0.45

The server sends a packet acknowledging the second and third parts of the request, and containing the first 512 bytes of the response. It follows this with another packet containing the second 512 bytes. The client then sends a message acknowledging the first two response packets.

```
.45116 ncsa.80 > unc.32840: . 1:513(512) ack 1147 win 8190
.45473 ncsa.80 > unc.32840: . 513:1025(512) ack 1147 win 8190
.45492 unc.32840 > ncsa.80: . ack 1025 win 9112 (DF)
```

Stage 5: Time = 0.53

The server sends the third and fourth response packet. The fourth packet also contains a flag indicating that the connection is being closed. The client acknowledges the data, then sends a message announcing that it too is closing the connection. From the point of view of the client program, the transaction is now over.

```
.52521 ncsa.80 > unc.32840: . 1025:1537(512) ack 1147 win 8190
.52746 ncsa.80 > unc.32840: FP 1537:1853(316) ack 1147 win 8190
.52755 unc.32840 > ncsa.80: . ack 1854 win 9112 (DF)
.52876 unc.32840 > ncsa.80: F 1147:1147(0) ack 1854 win 9112 (DF)
```

Stage 6: Time = 0.60

The server acknowledges the close.

```
.59904 ncsa.80 > unc.32840: . ack 1148 win 8189
```

Very nice, but what does it all mean

This trace is quite revealing. It seems that HTTP spends more time waiting than it does transferring data. Before we take a look at what causes these delays, we'll calculate a couple of metrics that will help us work out what's going on.

One important metric is the Round Trip Time (RTT). This is the time taken to sent a packet from one end of the connection to the other and back. Look at the gap between stages one and two, and between stages five and six, we can see that in our example, the RTT is around 70 milliseconds.

Another important metric is the bandwidth of the connection. This is a measure of how many bits per second our connection can carry. We can establish a lower bound on the available bandwidth by noting how long it took to send one of the data packets.

The two data packets in stage four arrive with a gap of 3.57 ms; since each packet carries 512 bytes of

data, this puts a lower limit on the available bandwidth of about 1.15 Mbps (143,750 MBps).

Connection Establishment

TCP establishes connections via a three-way handshake. The client sends a connection request, the server responds, and the client acknowledges the response. The client can send data along with the acknowledgement.

Since the client must wait for the server to send its connection response, this procedure sets a lower bound on the transaction time of two RTTs.

Data transfer: Segments

When TCP transfers a stream data, it breaks up the stream into small segments. The size of each segment can vary up to a maximum segement size (MSS). Although the segment size can be negotiated (the UNC host advertises an MSS of 1460 bytes in the first packet), this negotiation is optional. For remote connections, the MSS defaults to 536 bytes.

Since the NCSA host did not advertise an MSS, the UNC host uses the default value of 536 for this session.

Data transfer: Windows and Slow Start

Instead of having to wait for each packet to be acknowledged, TCP allows the sender to send out new segments even though it may not have received acknowledgements for previous segments.

To prevent the sender from overflowing the receivers buffers, in each segment the receiver tells the sender how much data it is prepared to accept without acknowledgments. This value is known as the window size.

Although the window size tells the sender the maximum amount of unacknowledged data the receiver is prepared to let it have outstanding, the receiver cannot know how much data the connecting networks are prepared to carry. If the network is quite congested, sending a full windows worth of data will cause even worse congestion. The ideal transmission rate is one in which acknowledgements and outgoing packets enter the network at the same rate.

TCP determines the best rate to use through a process called Slow Start. With Slow Start, the sender maintains calculates a second window of unacknowledged segments known as the Congestion Window. When a connection first starts up, each sender is only allowed to have a single unacknowledged segement in transit. Every time a segment is acknowledged without loss, the congestion window is opened; every time a segement is lost and times out, the window is closed.

This approach is ideal for normal connections; these connections tend to last a relatively long time, and the effects of slow start are negligible. However, for short lived connections like those used in HTTP, the effect of slow start is devestating.

How Slow Start affects HTTP

HTTP is hurt by slow start on both the client and server sides. Because the HTTP headers are longer than the MSS, the client TCP needs to use two segments (Stage 2). Because the congestion window is initialised to one, we need to wait for the first segment to be acknowledged before we can send the second and third (Stage 3). This adds an extra RTT onto the minimum time for the transaction.

When the server is ready to send the response, it starts with a congestion window set to 2. This is because the acknowledgement it sent in Stage 3 counts is counted as a successful transmission, allowing the congestion window to open up a notch before it comes time to send data.

Although the servers congestion window is already slightly open, it still doesn't have enough a big enough gap to send the entire response without pausing. In Stage 4, the server sends two segments carrying a combined 1K of data, but then needs to wait to receive the acknowledgement from the client before it can send the final two segments in Stage 5.

Since most pages are larger than 1K in size, slow start in the server will typically add at least one RTT to the total transaction time. Longer documents may experience several more slow start induced delays, until the congestion window is as big as the recievers window.

Readers should note that if the server were to support the negotition of larger MSS, this particular transaction would not have incurred any slow start delays. However, typical document sizes will cause at least one slow start pause when sending the result.

Latency and Bandwidth

Latency and Bandwidth are the two keys to protocol performance. Latency, as measured by the RTT, is a measure of the fixed costs of a transaction and does not change as documents get bigger or smaller. Bandwidth, as we discussed earlier, is a measure of how long it takes to send data.

Future Networks

Improving available bandwidth is relatively easy. All you need to do is buy a faster network. Reducing latency is somewhat harder- there are only two known ways to achieve this; either change the speed of light, or move. Since placing workstations in particle accelerators has been known to cause data loss, and since the second approach is in general contrary to the philosphy of distributed systems, latency is increasingly becoming the dominant factor in protocol performance.

Effects of opening new connection per transaction

The cost of opening a new connection for each transaction can be clearly seen by estimating the transaction time as it would have been if we had been reusing an existing connection.

In our example, the total transaction time was 530 ms

If were reusing an existing connection the transaction time could be calculated as the time to send the request plus the round trip time, plus any processing time on the server, plus the time to send the response. The only figure in this list for which we have no estimate is that for server processing time. This depends heavily upon machine load, but for our purposes, we can take the gap between sending the request and recieving the response, and subtract the response time (Stages 3 and 4). In this case, the approximate value is about 30 ms.

Using the above method, the expected transaction time for an already open server would be (1,146 / 143,750) + 0.07 + 0.03 + (1854 / 143,750), or abour 120 ms.

Effects Of Requesting A Single Object Per Transaction

Because HTTP has no way to ask for multiple objects with a single request, each fetch requires a single transaction. With the current protocol, this would require a new connection to be set up; however, even if the connection were to be kept open, each request/response pair would incur a separate round trip delay.

As an example, lets consider fetching ten documents, each the same size as the one we've been considering. For HTTP/1.0, the total time would be 5,300 ms. For the long-lived connection, the total transaction time will be 1,200 ms.

Ignoring any possible savings in header size and processing time, if we were to combine all the requests and responses into a single message, the total transaction time would be 10 * (3000/143,750) + 0.07 + (0.03 * 10), or 580 ms. If we assume that the optional headers are only sent once, then the total time per transaction is 520ms. In this model, processing time is the dominant factor. If we further assume that a proportion of processing time is independent of the number of objects requested, then the total time can be as low as 250ms (150ms transfer, 70ms latency, 30ms processing time).

On a gigabit network, the transfer time becomes insignificant. On such a network, the total time for the sequence of HTTP/1.0 requests would be approximately 5,100ms. If multiple objects per request were allowed over an already open channel, the total time would be 100ms.

Other Problems

One scalability problem caused by the single request per connection paradigm occurs due to TCP's TIME_WAIT state. When a server closes a TCP, connection, it is required to keep information about that connection for a period of time, in case a delayed packet turns up and sabotages a new incarnation of the connection.

The recommended time to keep this information is 240 seconds. Because of this, a server will leave some resources allocated for every single connection closed in the past four minutes. On a busy server, this can add up to thousands of control blocks.

Summary

HTTP/1.0 interacts badly with TCP. It incurs frequent round-trip delays due to connection establishment, performs slow start in both directions for short duration connections, and incurs heavy latency penalties due to the mismatch of the typical access profiles with the single request per transaction model. HTTP/1.0 also requires busy servers to dedicate resources to mainting TIME_WAIT information for large numbers of closed connections.

Bibliography

http
: HTTP protocol specification, Tim Berners-Lee, CERN

tcp
: "Transmission Control Protocol," J. Postel, RFC-793, September 1981.

slow-start
: "Congestion Avoidance and Control," V. Jacobson, ACM SIGCOMM-88, August 1988.