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 Section 4.3. Because TCP is a complex topic, we can provide only a brief overview of TCP performance here. Refer to Section 4.8 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.

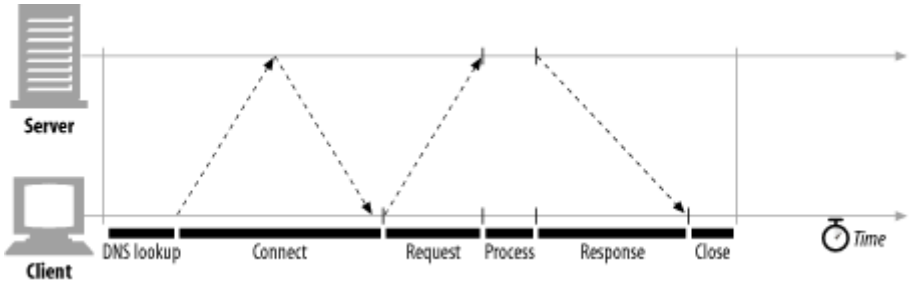

Figure 4-7. Timeline of a serial HTTP transaction

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. [3]

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.

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.

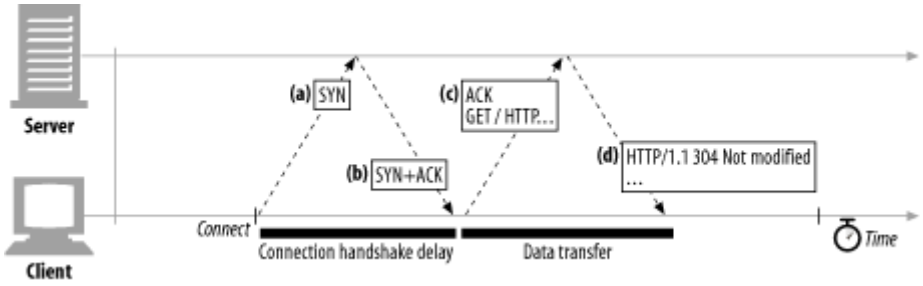

Figure 4-8. TCP requires two packet transfers to set up the connection before it can send data

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,[4] 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).

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 delayed 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.[5]

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.[6]

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 delayed 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.[7]

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.

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[8]), 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, destinatio
```

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 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.

[3] 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.

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

[5] Sending a storm of single-byte packets is called "sender silly window syndrome." This is inefficient, anti-social, and can be disruptive to other Internet traffic.

[6] 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.

[7] 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.

[8] 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.