8
HTTP/TCP Interaction

Although the Hypertext Transfer Protocol (HTTP) does not depend on any particular transport protocol, nearly every implementation of HTTP uses the Transmission Control Protocol (TCP). TCP was standardized in 1980, ten years before the emergence of the Web. Early application-layer protocols built on TCP differ markedly from HTTP. For example, Telnet is an interactive application that uses a single TCP connection to transfer data between a client and server over a period of time. In contrast, a Web client typically establishes multiple TCP connections to retrieve a collection of resources from a Web server. The File Transfer Protocol (FTP) maintains a single control connection between the client and server and transmits data on separate connections. In contrast, HTTP uses a single connection for transferring control and data. Compared with the files transferred by FTP, most Web request and response messages are relatively short. These unique characteristics of Web traffic have important implications for the efficiency of TCP.

This chapter discusses the interaction between HTTP and TCP and the implications for Web performance. First, we discuss how TCP implementations use the expiration of timers to trigger many key operations, such as the retransmission of lost packets. Although these timers affect any application-level protocol built on top of TCP, the characteristics of HTTP traffic lead to more dramatic performance effects on the Web than on earlier Internet applications. Next we explore how the separation of functionality between the transport and applications layers influences Web performance. Certain TCP mechanisms were motivated by earlier application-level protocols, such as Telnet and Rlogin. These features interact in subtle, and often negative, ways with HTTP. Next we discuss the performance and fairness implications of Web clients that have multiple TCP connections to the same Web server at the same time. For example, a browser may establish multiple connections to download multiple embedded images in a Web page. Busy Web servers must handle a large number of simultaneous TCP connections to a collection of different clients. We discuss ways to reduce the overhead on Web servers of handling a large number of TCP connections.

8.1 TCP Timers

TCP implementations rely on timers to trigger key protocol operations, such as the following:

- **Retransmission of lost packets:** The expiration of the retransmission timer triggers a TCP sender to retransmit a (presumably) lost packet.
- **Repeating the slow-start phase:** Some TCP implementations force a TCP sender to repeat the slow-start phase of congestion control after a period of inactivity.
- **Reclaiming state from a terminated connection:** The TCP sender that initiates the closure of the connection removes the state associated with the connection after a period of time has elapsed.

In this section, we explain why the timers that control these three operations have a significant influence on Web performance. Later, in Section 8.2.3, we discuss another timer that controls the transmission of delayed acknowledgments.

8.1.1 Retransmission timer

Web downloads sometimes stall for several seconds in a row, typically at the beginning of the transfer. These delays often stem from the time required for the TCP sender to detect that an Internet Protocol (IP) packet has been lost. In this section, we explain how the creation of the TCP connection can be delayed for several seconds as a result of a large initial retransmission timeout (RTO) value. Then we explain why retransmission timeouts occur relatively often in the middle of HTTP transfers, compared with other Internet applications.

Delay in establishing a TCP connection

From the user's viewpoint, clicking on a hypertext link translates directly into the display of the Web page in the browser window. Transparent to the user, the Web browser proceeds through several steps to retrieve the Web page—establishing a TCP connection, transmitting the HTTP request, receiving the HTTP response from the server, and rendering the resource. Because Web browsing is an interactive application, delay in any of these steps is visible to the user. This is in sharp contrast to noninteractive applications, such as the transfer of e-mail, in which the user does not expect an immediate reply. Although Telnet and FTP are interactive, these applications have a clear separation between establishing the connection and transferring the data. The typical user interacts with the remote machine for a relatively long period of time; therefore, a few seconds of additional delay in establishing the TCP connection and supplying a name and password do not necessarily have a considerable effect on the user's overall satisfaction.

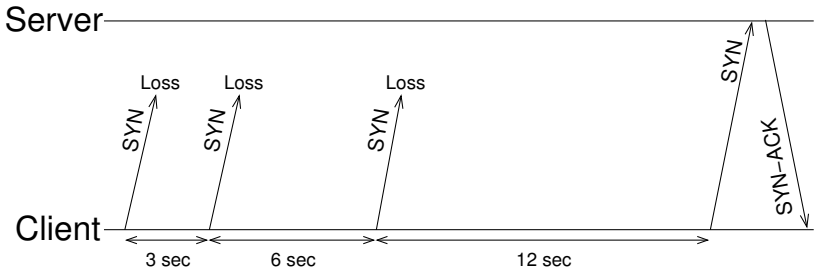

Figure 8.1. Client retransmitting lost SYN packet to server

Establishing a TCP connection requires a three-way handshake—a SYN from the client, a SYN-ACK from the server, and an ACK from the client. In the absence of packet loss, the client can transmit the HTTP request after one round-trip delay. However, the loss of the SYN or the SYN-ACK packet introduces a longer delay. TCP senders have two ways to detect a lost packet—duplicate acknowledgments and a retransmission timeout, as discussed in Chapter 5 (Section 5.2.5). However, the receiver does not send duplicate acknowledgments unless the TCP sender has transmitted multiple packets. At the beginning of the connection, a TCP sender has not transmitted any data packets, and the rate of packet transmission is limited by the small initial congestion window. In the absence of duplicate acknowledgments, the TCP sender must rely on the retransmission timer to detect a lost packet. Selecting the RTO value is a delicate process. A large RTO results in high latency in responding to lost packets, whereas a small RTO results in unnecessary retransmissions. To balance these trade-offs, the TCP sender selects an RTO based on estimates on the round-trip time (RTT) to the receiver.

However, at the beginning of a TCP connection, the sender has not accumulated any RTT measurements. This complicates the selection of an RTO for the initial packets of the connection. To address this problem, the TCP standard prescribes that the sender start with a default RTO of three seconds [PA00]. If the SYN-ACK packet does not arrive after the first retransmission of the SYN packet, the TCP sender increases the RTO from three seconds to six seconds and continues to double the timeout value after each successive retransmission, as shown in Figure 8.1. The combination of the large initial RTO and the exponential backoff avoids generating unnecessary retransmissions of the SYN and SYN-ACK packets when the hosts have a large RTT. If the SYN or SYN-ACK packet is lost, delaying the retransmission avoids overloading the already congested network. The conservative policy for retransmitting lost SYN and SYN-ACK packets does not have much effect on noninteractive applications.

Losing one or more SYN or SYN-ACK packets has a significant influence on Web performance. The loss of two successive SYN packets would result in a total delay of nine seconds before the transmission of the third SYN packet. Arguably, the likelihood of losing one or more SYN or SYN-ACK packets should be quite small. However, the advent of the Web has resulted in substantial network congestion and, consequently, higher packet loss rates (e.g., 5% or more) [Pax99]. Despite the deployment of high-speed links in many parts of the Internet, congestion still occurs on critical links. The links that connect a company, university, or even an entire country to the Internet may be very congested, particularly during the busiest hours of the day. The links between different Internet service providers are often heavily congested. Even if the network is lightly loaded, a busy Web server may lose SYN packets. The operating system on the server machine maintains a queue of pending TCP connections. If this queue is full, new SYN packets are discarded.

The high delay affects the behavior of Web users. The frustrated user may terminate the request by clicking on the Stop button on the browser. The user may then click on the Reload button to repeat the request. When the user clicks on the Stop button, the browser reacts by instructing the operating system to close the underlying TCP connection, as discussed in more detail in Section 8.2.1. When the user then clicks on the Reload button, the browser immediately initiates the creation of a new TCP connection. The operating system transmits a new SYN packet to the server and sets the RTO to its initial value (say, three seconds). If the new SYN packet reaches the server, then the TCP connection would be established much more quickly than waiting for a three-second or six-second timer to expire. The Web user's abort-and-reload behavior effectively triggers an immediate "retransmission" of the SYN packet. The user may also click on the Reload button without clicking on the Stop button. Clicking on the Reload button triggers both the termination of the existing TCP connection and the establishment of a new connection.

Stopping and restarting a stalled transfer typically reduces user-perceived latency, at the expense of higher load on the server and the network. For example, the operating system underlying a busy Web server may discard multiple SYN packets from different clients in a short period of time. If the users react by repeating their requests, each of these clients would send another SYN packet to the server. This exacerbates the already heavy load on the server machine. Similarly, user behavior can inflate the load on a congested network link. A heavily loaded link results in lost packets. Because most Web transfers are short, a relatively large fraction of the lost packets are SYN or SYN-ACK packets. If the lost packet causes the user to stop and restart the request, the client transmits a second SYN packet in a short period of time. This increases the amount of traffic that travels over the already congested link.

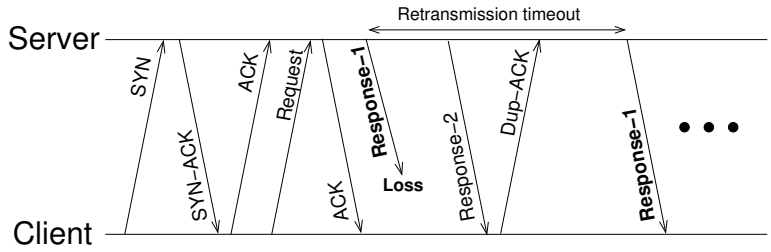

Figure 8.2. Client sending a single duplicate ACK to the server

Delay in the middle of a Web transfer

Compared with the beginning of a connection, long retransmission timeouts are less likely to occur in the middle of transferring a Web response for two reasons:

- **Smaller RTO value:** The TCP sender gradually refines the RTO value by observing the delay experienced by data packets. Over time, the RTO value becomes closer to the actual RTT between the sender and receiver. Consider a sender and a receiver with an RTT of 200 ms. At the beginning of the connection, the TCP sender has a three-second RTO. The TCP sender selects the RTO based on the average and variance of the RTT. Eventually, the RTO may drop to 250 or 300 ms.
- **Duplicate acknowledgments:** Retransmission timeouts are less likely to occur once the TCP sender starts transmitting data. As the data packets arrive, the TCP receiver sends acknowledgment packets that reflect the number of contiguous bytes that have arrived so far. Following a packet loss, the acknowledgment number does not increase for subsequent ACK packets. Upon receiving these duplicate ACK packets, the sender infers that the earlier data packet has not reached the receiver. The reception of three duplicate ACKs triggers the TCP sender to retransmit the missing packet without waiting for the retransmission timer to expire, as discussed in Chapter 5 (Section 5.2.5).

Both of these positive effects become more significant as the TCP sender transmits more data. The RTT estimate becomes more accurate as the hosts exchange more data, and the likelihood of duplicate acknowledgments increases as the TCP sender achieves a larger congestion window.

However, most Web responses involve a relatively small amount of data, in the range of 8 to 12 KB. These short transfers spend most, if not all, of their time in the slow-start phase of congestion control. The slow-start phase begins with a small initial congestion window of one or two packets, as discussed in Chapter 5 (Section 5.2.6). With a small congestion window, the likelihood of

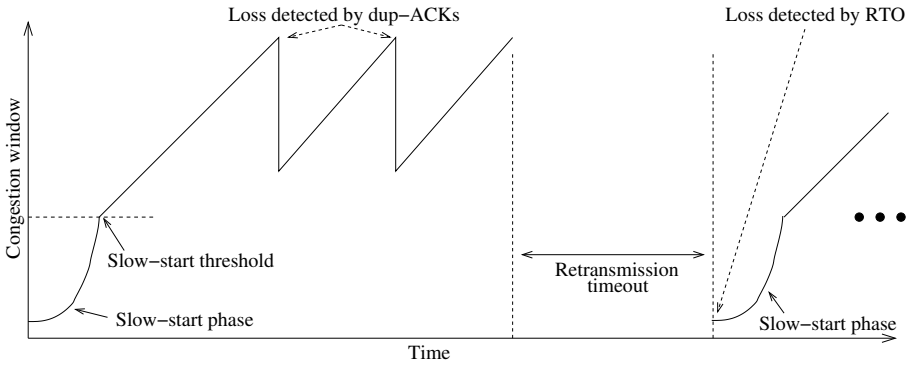

Figure 8.3. Repeating the slow-start phase after a retransmission timeout

successfully delivering multiple packets after a loss is very low. The TCP receiver is unlikely to generate the three duplicate ACKs necessary to trigger a fast retransmission of the lost packet [BPS+98]. For example, the loss of the first packet of a response message would result in a single duplicate acknowledgment triggered by the reception of the second packet, as shown in Figure 8.2. The small congestion window precludes the server from transmitting any additional packets until receiving an acknowledgment for the first packet. Waiting for the retransmission timer to expire stalls the transfer of data from the server. In addition, retransmission timeouts force the TCP sender to reset the size of congestion window to its small initial value, as shown in Figure 8.3. Retransmission timeouts can result in significantly long delays for Web transfers. Stopping and restarting the transfer may result in a faster response, compared with waiting for the retransmission timer to expire.

8.1.2 Slow-start restart

Persistent connections offer an attractive alternative to establishing a separate TCP connection for each Web transfer. Avoiding the slow-start phase of TCP congestion control is one of the purported advantages of reusing an existing TCP connection, as discussed earlier in Chapter 7 (Section 7.5). However, sending a response message on a previously idle persistent connection generates a large burst of packets. To avoid overloading the network, the TCP specification requires a TCP sender to repeat the slow-start phase of congestion control after a period of inactivity. Several techniques have been proposed to avoid repeating the slow-start phase without overloading the network.

Idle persistent connections

A Web client often downloads multiple resources from the same Web site in a short period of time. For example, the user may download a Web page that contains multiple embedded images. The browser requests these images from the server after parsing the container HTML file. In addition, a user may click on hypertext links to browse through multiple pages at the same server. With persistent connections, these HTTP transfers can travel over a single TCP connection. The transfers may experience higher throughput by having a larger congestion window. Suppose a client requests an HTML file, followed by requests for four embedded images. The transfer of the HTML file starts with an initial congestion window of one or two packets. The congestion window grows as the transfer proceeds through the slow-start phase. By reusing the connection, the transfer of the embedded images benefits from the larger congestion window. In fact, the congestion window may continue to grow as the server transfers multiple responses.

However, using a large congestion window may overload the network if the persistent connection has been idle for a period of time. Consider a user who downloads a Web page from a server and, after spending five seconds reading the page, retrieves another page from the same server. Assume, for the time being, that the client and the server have both decided to retain the connection. Network congestion may have changed substantially during the five-second idle period. While this TCP connection was inactive, other TCP connections on the same path may have increased their congestion windows to consume the available bandwidth. Consider the simple case of a busy link that carries traffic for five active TCP connections with the same round-trip times. On average, each of the five connections has a throughput of around 20% of link capacity. If one of these connections is idle for several seconds, the other four connections would start sending more aggressively until they each consume around 25% of the capacity.

Allowing the previously idle connection to transmit at its old rate could introduce substantial congestion. The connection would immediately start sending data at a rate that would consume 20% of the link capacity, while the other four connections continue transmitting data at 25% of link capacity. This generates a large amount of additional traffic—20% above the link capacity. Many of the packets traversing the congested link would be lost. In fact, packet losses may force several of the connections to reduce their sending rates. After the connections lower their sending rates, the link may become underutilized. The underutilization persists while the connections gradually increase their congestion windows. In addition to degrading overall performance, sending too aggressively may lower the throughput of the aggressive connection as well. The network congestion may cause the connection to experience a high rate of packet loss, forcing the retransmission of one or more packets.

Repeating the slow-start phase

To avoid generating a sudden burst of packets, the TCP sender should transmit less aggressively after an idle period. The slow-start phase of congestion control was designed precisely to avoid sudden and unexpected bursts of network traffic. During the slow-start phase, the TCP sender increases its congestion window and attempts to estimate its fair share of the bandwidth on its path through the network. Allowing a previously idle connection to use a large congestion window is similar to allowing a new connection to start with a large initial congestion window. The burst of traffic could overwhelm the network links and cause high packet-loss rates. To avoid these performance problems, the TCP specification requires a connection to repeat the slow-start phase following an idle period. This is referred to as the *slow-start restart* mechanism.

A precise definition of the slow-start restart mechanism requires the clarification of two key issues: the beginning of an idle period and the resetting of the congestion window. The idle period starts once the TCP sender stops transmitting data and all previous data packets have been successfully acknowledged by the receiver. After receiving the last acknowledgment, the TCP sender sets a timer. Once the timer expires, the sender resets the congestion window to its initial value of one or two maximum-size packets. The TCP sender uses the current RTO value to define the duration of the idle period. That is, if the application does not introduce any new data during the RTO period, the underlying TCP sender resets the congestion window and repeats the slow-start phase of congestion control.

Slow-start restart would occur if a connection is idle for longer than a few RTTs—typically a few seconds, at most. Most automatically generated requests for embedded resources would arrive within this time interval. However, persistent connections may also handle requests generated by users as they browse through multiple Web pages at the same site. Users frequently introduce longer delays between successive requests. The characteristics of these interrequest times are discussed in more detail in Chapter 10 (Section 10.5.3). The larger spacing between user-generated requests can trigger the slow-start restart mechanism. Repeating the slow-start phase reduces the performance gains achievable with persistent connections. A server cannot fully exploit a high-bandwidth path to the client if successive requests must start with a small congestion window. However, repeating slow start is important for the overall health of the network.

Reducing the slow-start restart performance penalty

The slow-start restart mechanism prevents the previously idle TCP connection from generating a large burst of traffic that would congest the network. Several

techniques have been proposed to reduce or avoid the performance penalties associated with slow-start restart [Hei97]:

- **Disabling slow-start restart:** The Web server could disable the use of the slow-start restart mechanism, at the risk of allowing a previously idle connection to generate a large, unexpected burst of traffic. The risk is reduced if the server closes connections after a short idle period. For example, the server may close a connection that has been inactive for more than 15 seconds to limit the total number of connections, as discussed later, in Section 8.4.2.
- **Using a larger slow-start restart timeout:** The operating system underlying the Web server could be configured to use a larger timeout parameter for triggering slow-start restart. However, the larger the timeout parameter, the more likely it is that the connection's previous congestion window is inconsistent with the current status of the network.
- **Gradually decreasing the congestion window:** Rather than an all-or-nothing solution, the TCP sender could *gradually* decrease the congestion window during the idle period. The TCP sender could reduce the congestion window in proportion to the length of the idle period. This adaptive approach becomes more conservative with the increasing uncertainty about prevailing network conditions.
- **Pacing the transmission of packets:** To avoid generating a burst of packets, the TCP sender could pace the transmission of packets into the network. Suppose the congestion window allows the sender to transmit four packets. Transmitting all four packets back-to-back would generate a large burst of traffic. Instead, the sender could introduce some delay after each packet transmission. This would reduce the likelihood of overloading the network links on the path to the receiver, while still allowing the sender to use the four-packet congestion window.

The TCP implementation on the Web server machine could employ a combination of these techniques. For example, the TCP sender could use a larger timeout, gradually decrease the congestion window, and pace the transmission of packets.

8.1.3 The TIME_WAIT state

Busy Web servers handle a high rate of requests for new TCP connections. Having a large number of TCP connections consumes memory resources and introduces overhead in processing incoming packets. Ideally, the operating system could reclaim these resources as soon as the connection closes. However, TCP requires one of the two hosts to retain information about the closed connection for a period of time. In this subsection, we explain why one host must enter the so-called TIME_WAIT state and why this responsibility often falls

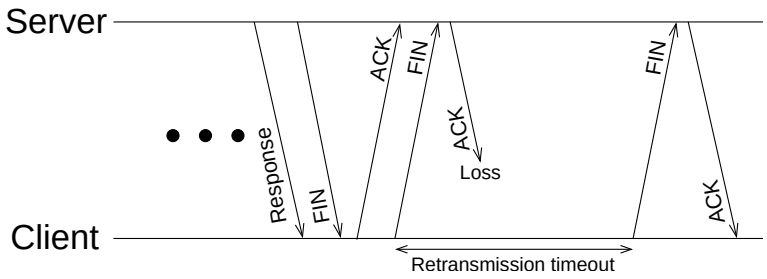

Figure 8.4. Loss and retransmission of final ACK packet from the server

upon the Web server rather than the client. Then we discuss several proposals for reducing the overhead of the TIME_WAIT state on Web servers.

Retaining information about closed connections

Retaining information about closed TCP connections introduces significant overhead on a busy Web server. Consider a Web server that closes a connection after sending an HTTP response to the client. Closing the connection initiates a four-way handshake, starting with the transmission of a FIN packet, as discussed in Chapter 5 (Section 5.2.3). Upon receiving the FIN packet, the TCP implementation at the client sends an acknowledgment packet. Then the client reads the HTTP response. After the last byte of the response message, the client reads an end-of-file (EOF) character that signifies that the server has closed its end of the connection. The client then closes its end of the connection, which triggers the transmission of a FIN packet. Upon receiving the client's FIN, the TCP implementation at the server sends an ACK, completing the handshake. After receiving the ACK, the client knows that the connection has been terminated. However, the *server* has no way of knowing if the client ever received the ACK packet.

Suppose that the server's final ACK packet was lost in the network, as shown in Figure 8.4. If the ACK packet was lost, the client would eventually experience a retransmission timeout. After the timeout, the client would send its FIN packet again. If the retransmitted FIN packet is lost, the client would experience another timeout and send the FIN packet again. Because the server does not know whether or not its ACK has reached the client, the server does not know if the ACK packet must be transmitted again. In the meantime, the server cannot delete its information about the TCP connection. Suppose that the server removes all information about the connection, under the false assumption that the client has received the ACK packet. Later, when a retransmitted FIN packet arrives, the server would not know that this packet belonged to the closed connection. Thinking that the FIN packet was sent erroneously,

the server would send an RST (reset) packet back to the client, rather than retransmitting the ACK packet.

A more serious situation arises when the connection has one or more outstanding packets in the network. Suppose that the server has received the entire request message and the client has received the entire response message. A duplicate (retransmitted) packet from this connection may remain in the network. That is, a packet may have been sent more than once, with one copy reaching the recipient and the other copy still traveling through the network. This packet may eventually reach the receiver. The handling of the duplicate packet depends on the status of the connection between the two hosts, as follows:

- **Connection is still open:** Suppose the duplicate packet arrives while the TCP connection is still open. The receiver inspects the sequence number of the packet and recognizes that this data has already been received. The receiver discards the duplicate packet.
- **Connection has been closed and no new connection exists:** Suppose that the connection has been closed and the two hosts have not established a new TCP connection. When the packet arrives, the operating system on the receiving machine inspects the port numbers in the TCP header to identify the connection associated with the data. No active connection matches these port numbers, so the receiver discards the packet.
- **Connection has been closed and a new connection exists:** Suppose that the connection has been closed and the two hosts have established a new connection using the *same pair* of port numbers. When the duplicate packet arrives, the port numbers are inspected by the operating system on the receiving machine. The port numbers match an existing connection, and the data is directed to the application associated with the new connection. This is a mistake, because the duplicate packet does not actually belong to this new connection.

To prevent the receiver from associating the duplicate packet with the wrong application, the two hosts must have a way to avoid creating a new connection with the same port numbers, at least for some period of time after the closure of the previous connection.

To prevent the establishment of a new connection with the same port numbers, at least *one* of the hosts must remember that the previous connection existed. The TCP specification assigns this responsibility to the host that sends the first FIN packet. On this host, the TCP connection enters the TIME_WAIT state. The operating system maintains information relating to the connection to retransmit the final ACK packet, if necessary, and to prevent creation of a new connection with the same IP addresses and port numbers. The connection must stay in the TIME_WAIT state long enough that no outstanding packets remain in the network. This is very difficult because IP does not provide a

limit on the worst-case delay for delivering a packet. In practice, the time-to-live (TTL) field in the IP packet header should ensure that a packet does not remain in the network indefinitely, as discussed in Chapter 5 (Section 5.1.4). The 8-bit field permits a maximum TTL value of 255 seconds.

TCP requires the host to remain in the TIME_WAIT state for twice the *maximum segment lifetime* (MSL), an estimate of the worst-case delay. The TCP standard specifies that implementations should use an MSL of 2 minutes [J. 81], though common implementations use 30 seconds, 1 minute, or 2 minutes. On the one hand, a small MSL reduces the duration of the TIME_WAIT state, which reduces the amount of system resources devoted to retaining information about closed TCP connections. However, a small MSL also increases the likelihood that a duplicate packet remains in the network. Likewise, a small MSL increases the likelihood that the sender is unable to retransmit a lost final ACK packet. On the other hand, a large MSL results in a long stay in the TIME_WAIT state. This results in a potentially large number of connections in the TIME_WAIT state at the same time, which could consume significant system resources on a busy Web server. In addition, a large MSL value could actually limit the achievable rate of communication between the two hosts.

Suppose a client creates a TCP connection with port 1025 to a server running on port 80 on a remote host. After sending the FIN packet to close the TCP connection, the server enters the TIME_WAIT state for four minutes. After receiving the ACK from the server, the client terminates the connection. The client may wish to establish another connection to the same server to request another resource. This new connection must use a different client port (such as 1026) because the server is still in the TIME_WAIT state for the old connection. The TCP header has a 16-bit field for the port number, which restricts the number of different ports that can be used by the client during the four-minute period. In practice, this would not impose a significant limitation on a typical Web browser. However, a busy proxy may need to establish TCP connections to a popular server at a very high rate. The restriction is even more significant if the proxy is configured to send all requests to a downstream proxy. The inability to establish new connections with the old port numbers limits how often the proxy can send requests to the next proxy in the chain.

Effect of TIME_WAIT on Web servers

Web servers often initiate the closure of a TCP connection and bear the overhead associated with the TIME_WAIT state. First, consider the case in which the client and the server do not maintain a persistent connection. In this situation, the server would typically close the connection (by sending a FIN packet) immediately after sending the HTTP response to the client. Second, consider the case in which both the client and the server keep the TCP connection open.

In this scenario, eventually either the client or the server closes the TCP connection. The server often has a stronger incentive to close the connection. For example, a busy Web server cannot afford to maintain a persistent connection for every client. In fact, a Web server typically applies an application-level timeout to close a TCP connection after a period of inactivity, as discussed later, in Section 8.4.2. In contrast, a Web browser may have little incentive to close a persistent connection because reusing the existing connection would reduce the latency experienced by the user over establishing a new connection.

In either of these two scenarios, the server closes the connection first and hence must enter the TIME_WAIT state. The situation becomes more complicated when the client is an intermediary, such as a proxy. A busy proxy has connections to a large number of clients and servers. A proxy may have as much of an incentive to close the connection as the server, if not more. If the proxy closes the connection, then the proxy incurs the burden of the TIME_WAIT state, rather than the server. The TIME_WAIT state imposes a heavy burden on busy proxies as well. In general, the TIME_WAIT state introduces an unfortunate trade-off—because busy hosts have an incentive to close connections, the busy hosts must bear the overhead of the TIME_WAIT state.

The overhead of the TIME_WAIT state is exacerbated by the fact that the TCP connections are relatively short-lived because most Web responses are small. Suppose a client requests a single resource from a Web server. The server may spend a few seconds sending the response message. After closing the connection, the server must stay in the TIME_WAIT state for four minutes. This is an extremely long time relative to the period that the connection was open. Earlier Internet applications, such as Telnet and FTP, typically used TCP connections for a longer period of time. Compared with an FTP server, a Web server has a larger proportion of its TCP connections in the TIME_WAIT state. The use of persistent connections partially addresses the problem. Using a single TCP connection for multiple HTTP transfers reduces the total number of TCP connections opened, and closed, at the server. This, in turn, reduces the number of TCP connections in the TIME_WAIT state. However, a busy Web server may still have a large number of connections at a time in the TIME_WAIT state.

Reducing TIME_WAIT overhead

Reducing the burden of the TIME_WAIT state is extremely important for high-performance Web servers. Techniques for reducing this overhead fall into two main categories—lowering the system resource requirements for TIME_WAIT connections and shifting the responsibility for the TIME_WAIT state to Web clients. The operating system must retain information about each connection in the TIME_WAIT state. Storing this information consumes memory that could be used for other purposes, such as caching frequently accessed Web resources.

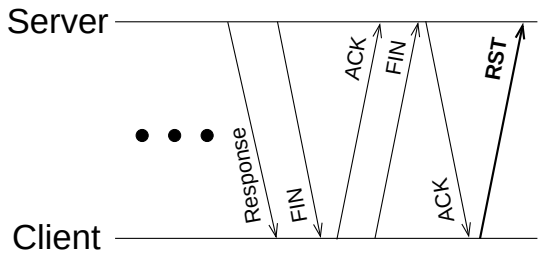

Figure 8.5. Client sending RST to free server from the TIME_WAIT state

Fortunately, the operating system does not need to retain as much information for TIME_WAIT connections as for active connections. Modern operating systems reduce the memory requirements to the bare minimum necessary for the operating system to retransmit the final ACK packet and to prevent the creation of a new connection with the same IP addresses and port numbers. Having a large number of connections in the TIME_WAIT state also increases the overhead for the operating system to check for expired TCP timers, such as the retransmission timer. Most operating systems check for expired timers by periodically scanning the list of TCP connections. Placing the TIME_WAIT connections at the end of this list reduces the processing time required to scan the list. Reducing the memory and processing requirements for TIME_WAIT connections results in a substantial increase in the throughput of busy Web servers [AD99].

Despite the benefits of reducing the memory and processing requirements, the TIME_WAIT state still imposes an overhead on Web servers. Several techniques have been proposed that shift the responsibility for the TIME_WAIT state to the client [FTY99], which presumably handles a relatively smaller number of TCP connections, as follows:

- **Change TCP:** The specification of TCP could be changed to have the *recipient* of the first FIN packet incur the burden of the TIME_WAIT state. For example, suppose the Web server closes a TCP connection and sends a FIN packet to the client. After receiving the server FIN, the client TCP implementation could transition to the TIME_WAIT state. To ensure that the server does not enter the TIME_WAIT state, the client could send an RST packet to the server, as shown in Figure 8.5. Receiving an RST packet triggers the operating system at the server to leave the TIME_WAIT state and reclaim the memory resources associated with the TCP connection. An alternative approach involves changing the procedure for establishing a TCP connection to require the two hosts to negotiate who should handle the TIME_WAIT state upon connection closure.

- **Change HTTP:** The specification of HTTP could be changed to have the client initiate the closure of the TCP connection. For example, a new response header could allow the server to instruct the client to close the connection after receiving the complete response message. A new request header could allow the client to express its willingness to assume responsibility for closing the connection. For example, the client could close the TCP connection after automatically downloading the embedded images in a Web page. However, relinquishing control to the client may not be an appropriate solution in practice. The client may not have an incentive to close the connection, or to assume responsibility for the TIME_WAIT state.

However, each of these approaches requires changes to TCP or HTTP. Neither of the proposed approaches offers an attractive solution to the performance problems associated with the TIME_WAIT state. Instead, administrators of busy Web servers typically modify the operating system to reduce the TIME_WAIT timeout (e.g., five seconds). This reduces the TIME_WAIT overhead, at the risk of establishing a new TCP connection with the same port numbers.

8.2 HTTP/TCP Layering

The transfer of Web resources draws on the application-layer functions provided by HTTP and the transport-layer functions provided by TCP. In some cases, it is not clear which layer should perform a certain function. This section discusses three examples in which functions implemented at the transport layer have a significant effect on Web performance:

- **Aborted HTTP transfers:** Because the HTTP protocol does not have a mechanism for terminating an ongoing transfer, aborting an HTTP request requires closing the underlying transport connection, as discussed in Section 8.2.1.
- **Nagle's algorithm:** Nagle's algorithm limits the number of small packets transmitted by a TCP sender, which may delay the transfer of the last packet of an HTTP message, as discussed in Section 8.2.2.
- **Delayed acknowledgments:** The TCP receiver may delay transmission of an acknowledgment in the hope of piggybacking the acknowledgment on an outgoing data packet, at the expense of increasing the latency in transferring an HTTP message, as discussed in Section 8.2.3.

Handling aborted transfers at the transport layer avoids the need for HTTP to have an intricate relationship with any particular transport protocol. Nagle's algorithm and delayed acknowledgments were included in implementations of

TCP to reduce the overhead of transferring data for interactive applications such as Telnet and Rlogin.

8.2.1 Aborted HTTP transfers

A user may abort an ongoing Web transfer by clicking on the Stop button or by clicking on a hypertext link to retrieve another page. Abort operations are a common part of conventional Web browsing. In some cases, the browser may display the contents of a page as the response message arrives from the server. This allows the user to read part of the page and, perhaps, click on a hypertext link before the page has arrived in its entirety. In particular, the user may click on a hypertext link before all of the embedded images have been downloaded. In other cases, a user may interrupt a transfer that is proceeding slowly and then hit the Reload button to attempt to download the page again. A user with slow network access, such as a telephone modem, may be more likely to abort Web transfers. Transfers of large resources are likely to be aborted before they complete. In this subsection, we explain why aborting a Web transfer requires closing the underlying transport connection and discuss the implications on Web performance.

Absence of an abort mechanism in HTTP

HTTP does not provide a way for the client to communicate its desire to terminate the ongoing transfer. The notion of aborting a request lies at the boundary between the transport and application layers. Including an abort mechanism in HTTP would have been difficult to do without specifying the interaction with the underlying transport protocol. Although Web transfers typically employ TCP, the HTTP specification is divorced from the details of any particular transport protocol. Other application-level protocols, such as Telnet, have faced the same challenges. Telnet chose a different solution. During a Telnet session, a user can type cntl-C or the Delete key to abort the current command. The sender can use the urgent pointer field in the TCP header to draw the immediate attention of the receiver, as discussed in Chapter 5 (Section 5.2.7). This avoids the need for the receiver to process the previous bytes of the stream. However, this optimization is tied closely to TCP.

HTTP/1.1 could have followed a similar approach by introducing an abort request method. However, introducing an abort request would complicate the handling of pipelined requests. Suppose that a client has sent several pipelined requests on a persistent connection, followed by an abort request on the same connection. The server would need to read ahead in the list of pipelined requests to learn that the client has requested an abort. If the client pipelines several requests, either the server would have to abort all of the pending requests or the client would need to indicate which requests should be aborted. Addressing

these problems would have introduced substantial complexity to the protocol. Thus the specification of HTTP/1.1 does not include an abort mechanism.

Consequently, a Web client has no effective way to abort an ongoing HTTP transfer short of terminating the underlying transport connection. Consider a user that aborts the transfer of a 20 MB file after 5 MB have arrived. The client has two choices: terminating the TCP connection or receiving 15 MB of unnecessary data. Retrieving extra data imposes a burden on the network and consumes bandwidth that could be used to satisfy the user's next request. This increases user-perceived latency. Hence, most Web client implementations choose to terminate the TCP connection.

Effect of abort operations on Web performance

Aborting the TCP connection has important implications for Web performance. Consider an HTTP/1.1 browser that has a persistent connection to an HTTP/1.1 server. After retrieving an HTML file, the browser generates a series of pipelined requests for the various embedded images. Suppose that while the server is transmitting these images, the user clicks on a hypertext link to access another Web page at the same site. The browser terminates the TCP connection to abort the transfer of the embedded images. Terminating the TCP connection avoids the transfer of the remainder of the embedded images. However, the user must wait for the browser to establish a new TCP connection to the server before the next Web page can be retrieved. This requires the typical three-way handshake to open the new connection and the repetition of the TCP slow-start phase.

Aborted transfers can introduce additional problems when the HTTP request has a side effect. For example, the user's request could create a new resource, increment a variable, or trigger a script that purchases a product at an e-commerce site. If the TCP connection is terminated before the browser receives the server's response message, then the user does not know whether the request was completed at the server or not. HTTP does not provide any way for the server to unilaterally contact the client to indicate whether the request was processed. The Web site may maintain additional state that enables the client to determine whether the previous request was processed. Suppose a user visits an e-commerce site and sends a request that would add an item to a virtual shopping basket. Upon aborting the request, the user may not know if the item was successfully added to the basket. The Web site may allow the user to visit a Web page that displays the current contents of the shopping basket. This would allow the user to determine whether the previous request had completed before the TCP connection was aborted. Alternatively, the HTML form could include a unique transaction number that enables the script processing the request to recognize whether the transaction has already been performed.

Abort operations tighten the coupling between pipelined requests on the same TCP connection. Aborting one request in the pipeline requires aborting all of the pipelined requests that have not been processed yet. For example, aborting the downloading of a Web page terminates the transfer of all of the embedded images. This captures the intent of the user to abort the transfer of the entire page, rather than the downloading of a single Web resource. However, consider the case of a proxy that pipelines requests from two clients on a single persistent connection to a Web server. The proxy issues a request on behalf of client A, followed by a request on behalf of client B. If client A aborts its request, then the proxy has two choices. The proxy could keep the connection open and receive the server's entire response to the first request. This is wasteful because client A does not want to receive this data. Alternatively, the proxy could abort the connection. This would require the proxy to open a new connection and send client B's request a second time. Neither option is attractive. The proxy cannot avoid this problem without having a separate TCP connection to the server on behalf of each client.

The user-level abort operation does not immediately stop the transfer of data from the server. Consider a browser communicating directly with a Web server. When the user clicks on Stop, the browser initiates termination of the TCP connection by sending a FIN or RST packet to the server. In the meantime, the server continues to send data to the client. Depending on the propagation delay and the size of the Web response, the entire response may be transmitted before the server receives the RST/FIN packet. The problem is exacerbated when the client sends the request via an intermediary, such as a proxy. In this case, the HTTP transfer involves a TCP connection between the client and the proxy and a second TCP connection between the proxy and the server. Suppose that the server-proxy path has high bandwidth relative to the proxy-client path. This is a common scenario when the client has a low-speed modem. Because of the mismatch in network bandwidth, the transfer between the server and the proxy may proceed much more quickly than the transfer between the proxy and the client.

Consider a client that requests a 20 MB resource and aborts the transfer after receiving 5 MB of the response. The proxy may have received the entire resource from the server. The transfer of the last 15 MB is wasted, unless the proxy receives another request for the resource in the near future. If the client had communicated with the server over its slow connection, rather than through a proxy, the server would not have sent the additional 15 MB. Flow control on a single TCP connection between the client and the server would have prevented the server from sending so aggressively. Preventing the transfer of excess data requires some coupling between the server-proxy and proxy-client connections. The proxy could limit how much data it reads from the connection to the server. By not reading the data in the receive buffer, the proxy reduces the receiver window of the TCP connection. Reducing the receiver window size limits how

much data the server can transmit. Deciding how much data to read from the receive buffer introduces a fundamental tension between avoiding excess traffic for aborted transfers and reducing latency for normal transfers.

Details of aborting the TCP connection

The browser aborts an ongoing data transfer by invoking a system call to close the connection to the Web server. Depending on the browser implementation and the operating system, the system call may generate either a FIN or an RST packet. Assume that the system call triggers the transmission of a FIN packet. Upon receiving the FIN packet, the operating system on the server machine delivers an EOF to the server application. The operating system continues to transmit data from the send buffer to the remote client. After reading the EOF, the server stops writing new data into the send buffer. The data already in the send buffer and in the network would be delivered to the machine running the browser. Whether the browser application sees this additional data or not depends on how the connection was closed. If the browser has closed both the reading and writing ends of its connection, the operating system would discard the data. If the read direction of the connection remains open, then the operating system would deliver the data to the browser application. Continuing to receive data may be useful if the browser plans to cache the partial contents to satisfy future user requests.

Some UNIX implementations do not allow applications to initiate the transmission of an RST packet. For these systems, an abort would generate a FIN packet, whereas other operating systems may trigger an RST packet. Upon receiving an RST, the operating system on the server machine discards any remaining outgoing data for the connection, including data that the server application has already written into the send buffer. This is both good and bad. On the positive side, resetting the TCP connection avoids the transfer of additional data from the server. In addition, the RST packet causes the operating system on the server machine to discard any data residing in the receive buffer, rather than allowing the receiving application to read the data. This would obviate the need for the server to read any pipelined HTTP requests that might reside in the receive buffer. On the negative side, an RST does not close the connection in a clean manner. For example, suppose that some of the packets sent by the server machine had not reached the receiver. After receiving an RST, the operating system on the server machine would not retransmit these lost packets.

8.2.2 Nagle's algorithm

Interactive applications such as Rlogin and Telnet typically generate many small packets to transmit user keystrokes and short responses. Nagle's algorithm reduces the number of small packets by delaying the transmission of

data [Nag84]. After discussing the motivation for limiting the number of small packets, we describe how Nagle's algorithm degrades the performance of Web transfers, particularly under persistent connections. Then we explain how the Web server can prevent the transmission of small packets even when Nagle's algorithm is disabled.

Reducing the number of small packets

Consider the Telnet application that allows a user to interact with a machine in another location. This application coordinates the transfer of user keystrokes to the remote machine and the transfer of the machine's responses back to the user. Quick responses are necessary to give the user the illusion of interacting directly with the remote machine. However, the TCP sender underlying the Telnet application should not generate a separate IP packet for each keystroke. Otherwise, each keystroke would result in a 41-byte packet—a 20-byte IP header, a 20-byte TCP header, and 1 byte of data. Sending 41 bytes for every byte of data would introduce significant overhead, resulting in heavy congestion in the network. After receiving data from the application, the operating system should wait to accumulate additional data before transmitting a packet. Interactive applications are not tolerant of latency; therefore the operating system should not delay the transmission for very long. Nagle's algorithm addresses this trade-off. The algorithm ensures that the TCP sender transmits at most one small packet per RTT. In this context, a "small" packet is a packet containing fewer bytes than the maximum segment size (MSS) for the TCP connection (e.g., 536 or 1460 bytes).

Consider a TCP sender that has transmitted a packet and is waiting for an acknowledgment from the receiver. The TCP sender does not transmit any small packets until all outstanding acknowledgments are received. By this time, the sender may have accumulated additional data. On a wide-area connection with a large RTT, Nagle's algorithm avoids having multiple short packets in flight at the same time. On a local-area connection with a small RTT, the acknowledgment packets from the receiver almost always arrive before the sender has new data to transmit. Limiting the number of short packets in flight would not introduce any extra delay before sending the next data packet. In addition, Nagle's algorithm would not affect bulk-transfer applications because the transmission of a large file typically results in full-size segments. The approach has the most influence precisely when it is needed—for interactive applications communicating over a connection with a large RTT.

Nagle's algorithm and persistent connections

Nagle's algorithm can have a negative effect on Web performance when Web transfers result in the transmission of small TCP segments. Consider a Web

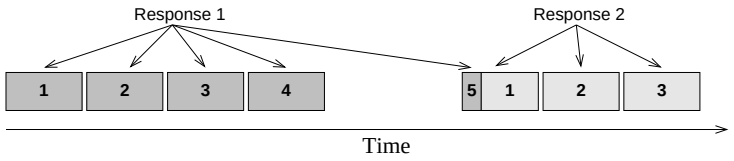

Figure 8.6. Server transmitting full-size packet containing end of response

server that transmits an HTTP response message by writing the response header followed by the response body, using two separate system calls. The underlying operating system might transmit the HTTP header in a single, small segment before the Web server has performed the second system call that writes the response body into the send buffer. Suppose the response body is also smaller than the maximum segment size. In theory, the operating system would have to delay the transmission of the response body, in the hope of accumulating more data. However, if the server application initiates the closure of the connection, the operating system does not expect the application to provide any additional data over the connection. This triggers the transmission of the small segment, regardless of whether the acknowledgment of the first packet has arrived. If the server does not close the connection, the operating system does not transmit the small second segment until the acknowledgment of the first segment arrives. In addition to the RTT for the acknowledgment to reach the server, the operating system at the client machine may delay the transmission of the acknowledgment packet, as discussed later in Section 8.2.3.

Even if the server writes the entire response message in one step, Nagle's algorithm can degrade the effectiveness of persistent connections [Hei97]. Consider a Web server that writes an HTTP response message into the send buffer. This would generate a relatively large amount (say, 8 to 12 KB) of data, typically in a short period of time. The operating system would transmit the response message as a series of full-size packets. At most, depending on the size of the response, there may be one small final packet at the end of the message. For example, consider a 6000-byte message that is transmitted over a TCP connection with an MSS of 1460 bytes. This would result in four 1460-byte segments and one 160-byte segment. The operating system would transmit the sequence of full-size packets and then delay the transmission of the small final packet. The transmission of the final packet is triggered by one of the following two events:

- **Writing of additional data into the send buffer:** The server application may write additional data into the send buffer, resulting in the transmission of a full-size packet, as shown in Figure 8.6. The packet contains the end of the first response message and the beginning of the second response message. However, the server does not necessarily have additional

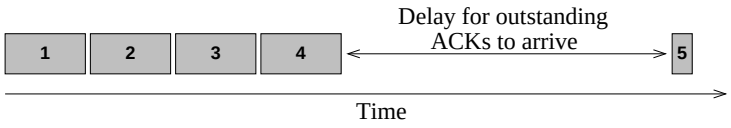

Figure 8.7. Server transmitting small packet after receiving ACKs

data to transmit to the client. For example, the server may not have additional requests from this connection to process at this time.

- **Receiving all outstanding acknowledgments:** Acknowledgments may arrive for all of the outstanding data that was transmitted to the receiver, allowing the operating system to transmit the small final packet. Figure 8.7 shows an example in which the server has an initial congestion window of two full-size packets and receives an ACK packet for every other data packet. Waiting for acknowledgments introduces an RTT delay. For small response messages, the round-trip delay in sending the small final packet may be a very significant part of the total latency.

Disabling Nagle's algorithm is an easy way to avoid these performance penalties. This is achieved by setting the appropriate option when establishing the connection. For example, the UNIX operating system has a *setsockopt()* function for setting options. Setting the TCP_NODELAY option in the *setsockopt()* call disables Nagle's algorithm. Web servers supporting persistent connections typically disable Nagle's algorithm. Web clients may also disable Nagle's algorithm. Otherwise, the transmission of large request messages, such as `PUT` and `POST` requests, may encounter performance penalties.

Disadvantage of disabling Nagle's algorithm

On the surface, disabling Nagle's algorithm should not have any undesirable effects on Web performance. However, Nagle's algorithm provides important protection when the application writes data in small increments. Consider a Web server that performs a separate system call to write each line of the HTTP response header. With Nagle's algorithm disabled, each *write()* call could result in a separate packet. This would be very inefficient. For example, consider a 300-byte response header consisting of ten lines. The header should fit in a single IP packet. In fact, the packet could also include the initial part of the response body, if one exists. However, writing the header lines one at a time would result in up to ten packets when Nagle's algorithm is disabled. This problem was common in the early NCSA Web server, which invoked a system call to write each response header. A similar phenomenon has also been observed in a Network News Transfer Protocol (NNTP) server [MSMV99].

The likelihood of having a separate packet for each header depends on the load on the server. A heavily loaded server is less likely to send a large number of small packets. Consider a busy server that is generating and sending hundreds of responses at the same time. The transmission of IP packets is limited by the bandwidth of the underlying network connection. During periods of heavy load, the operating system must queue the data written by the server. Consider a server process that invokes ten system calls to write the headers of a single response message. On a heavily loaded server, the second system call may occur before the server transmits the data from the first system call. The operating system combines the data from the two system calls into a single packet. In fact, the operating system may transmit all ten headers in a single packet. If the server were lightly loaded, the operating system would transmit the ten headers as separate packets. Each of these packets would incur a 40-byte overhead for the IP and TCP headers.

The server can avoid generating small packets without enabling Nagle's algorithm by generating the entire response header and then invoking a single *write()* call to write the header into the transmit buffer. This ensures that the operating system does not create separate packets for each header line and also avoids the overhead of performing multiple system calls. In fact, the server could conceivably perform a single system call to send both the HTTP header and the response body, as discussed in more detail later, in Section 8.4.1.

8.2.3 Delayed acknowledgments

By design, TCP supports bidirectional communication between two hosts. When both hosts transmit traffic, TCP acknowledgments can be piggybacked on data transfers. Delaying the transmission of an acknowledgment increases the likelihood of piggybacking the ACK on a data packet. Although effective for two-way applications such as Telnet and Rlogin, delaying the transmission of acknowledgments increases the latency for Web transfers. If Nagle's algorithm is enabled, delayed acknowledgments can stall the downloading of the last portion of a Web page. In this subsection, we discuss the motivation for delayed acknowledgments and examine the implications on Web performance.

Motivation for delayed acknowledgments

A TCP sender depends on acknowledgments from the receiver to pace the transmission of data. The congestion window controls the packet transmissions. Once the window is full, the sender cannot transmit new data packets before receiving an ACK from the receiver. Receiving ACKs in a timely manner is crucial to sustaining a high data-transfer rate. However, sending an ACK requires the receiver to transmit a 40-byte packet—a 20-byte IP header and a 20-byte TCP header, with the ACK bit set. This is very inefficient. Consider a sender that transmits 400-byte packets to the receiver. Sending a 40-byte acknowledgment

packet for every data packet would increase the amount of traffic in the network by 10%. The amount of acknowledgment traffic can be reduced in two ways. First, the receiver does not necessarily have to send an ACK for every data packet. Second, the receiver could piggyback the ACK information (the ACK flag and acknowledgment number) while sending data packets of its own.

Piggybacking the ACK on an outgoing data packet avoids sending separate acknowledgment packets. This is very effective when the receiver has data of its own awaiting transmission. However, piggybacking is impossible when the receiver does not have data to send. To increase the likelihood of piggybacking, TCP allows the receiver to *delay* transmission of the ACK packet in the hope that the application will generate data soon. This is very effective for interactive applications. Consider a user running on machine A that has an Rlogin session with machine B. Suppose the user types one or more characters. These characters are sent over the TCP connection to B. Then the Rlogin application reads these characters from the connection. The application generates an echo of these characters to be displayed on the screen on machine A. The echo generates a data packet to be sent from B to A. Ideally, B would acknowledge reception of the initial characters and send the echo with a single packet. However, this is not possible if B generates an ACK immediately after receiving the data packet from A. Instead, the receiver could delay the ACK and piggyback it on the packet with the echoed characters.

The longer B waits to send the ACK, the higher the chance of piggybacking on an outgoing data packet. However, B should not wait too long. Acknowledging the receipt of the packet from A is important. A may not be able to send additional data if the transfer is not acknowledged. Hence, delaying the ACK could increase the latency of transfers from A to B. To balance this trade-off, TCP invokes a timer to trigger the transmission of the ACK, even if no outgoing data is available. Most TCP implementations send outstanding ACKs every 200 ms, although some implementations introduce a delay of up to 500 ms. To avoid delaying ACKs for busy connections, TCP requires that at least every other full-size packet must be acknowledged, even if the delayed-ACK timer has not expired. For example, if a Web server is transmitting a large response message to a client at high speed, the client would transmit an ACK packet immediately after receiving every other data packet, even if the client did not have outgoing data to send to the server.

Interaction of delayed ACKs with HTTP traffic

Delayed ACKs reduce the amount of acknowledgment traffic in two ways: by piggybacking the ACK on an outgoing data packet and by sending an ACK for every other data packet, rather than every packet. Piggybacking of ACKs is very unlikely for Web traffic. A typical Web transfer involves a small HTTP request message from the client, followed by an HTTP response from the server.

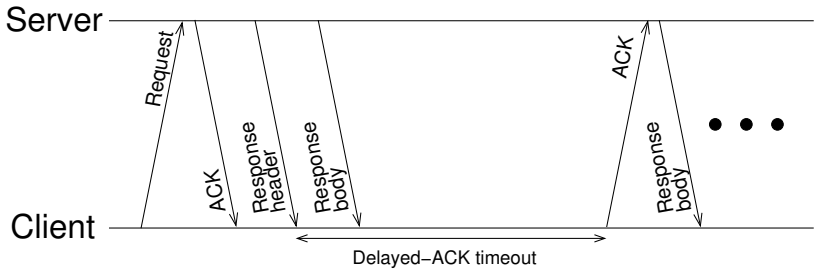

Figure 8.8. Client delaying the ACK of first two segments from server

It is very unlikely that *both* hosts would transmit data at the same time, except perhaps when a server is sending a response while the client is pipelining subsequent request messages. Delaying the transmission of an ACK packet rarely allows the client to piggyback the acknowledgment on an outgoing data packet. Instead, the 200 to 500 ms latency introduced by the delayed-ACK timer can degrade Web performance. This latency may be visible to the user. In addition, when the server's congestion window is full, delaying the transmission of acknowledgments stalls the transmission of the rest of the response.

The delayed-ACK mechanism can introduce unnecessary delay for Web traffic, depending on how the server software is written. Consider a Web server that transmits an HTTP response header and body in two separate steps. The HTTP response header is typically smaller than the MSS. The operating system may transmit the small packet before the application writes any additional data into the send buffer. Later, the server application starts writing the response body into the send buffer. If the server has disabled Nagle's algorithm, the operating system can start transmitting the packets containing the response body. Upon receiving the first full-size packet, the client has received two packets from the server. However, the client does *not* generate an ACK because the first packet (the HTTP header) was smaller than the MSS. The client must wait for the delayed-ACK timer to expire before sending an ACK. Depending on the congestion window size, the server may not be able to transmit additional data until the ACK arrives, as shown in Figure 8.8. To avoid this problem, some operating systems disable the delayed-ACK mechanism at the beginning of a connection. However, this does not prevent the problem from arising for subsequent response messages that use the same TCP connection.

Nagle's algorithm has a subtle interaction with the delayed-ACK mechanism when the server and the client maintain a persistent connection [Hei97]; the same phenomenon has been observed in an NNTP server [MSMV99]. Consider a Web server that has written an HTTP response into the send buffer. The operating system divides the response message into TCP segments, each

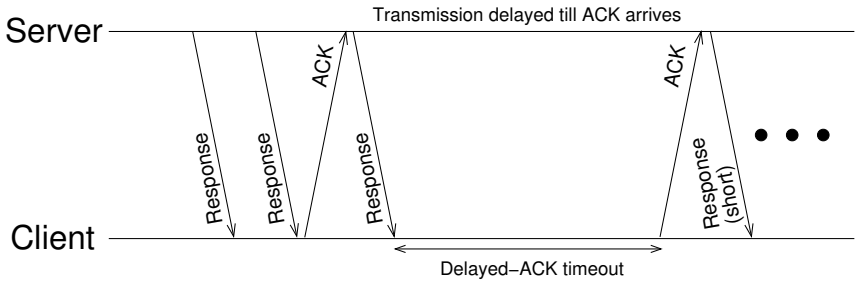

Figure 8.9. Interaction of Nagle's algorithm with delayed ACKs

transmitted in an IP packet. The operating system tries to send data in full-size packets, though the last packet is typically smaller than the others. For example, consider a 5000-byte message that is transmitted as three 1460-byte segments and one 620-byte segment. Following the delayed-ACK algorithm, the client acknowledges each pair of full-size packets. After receiving acknowledgments for the first two packets, the server transmits the third full-size packet. Upon receiving this packet, the operating system on the client machine does not transmit an acknowledgment packet, in accordance with the delayed-ACK mechanism. However, the operating system on the server machine does not transmit the final small segment until receiving all outstanding acknowledgments. The server transmission stalls awaiting an ACK from the client (because of Nagle's algorithm), and the client delays the acknowledgment (because of the delayed-ACK algorithm), as shown in Figure 8.9. Fortunately, disabling Nagle's algorithm at the server can prevent this situation from arising in practice.

8.3 Multiplexing TCP Connections

The previous two sections consider the dynamics of an individual TCP connection. However, a client often establishes *multiple* TCP connections to a server. Having multiple TCP connections at the same time has important performance and fairness implications. In this section, we examine the motivations for clients to establish multiple connections, discuss the resulting performance issues, and consider several ways to control the potential performance problems.

8.3.1 Motivation for parallel connections

A Web client has several incentives to have multiple TCP connections to a Web server at the same time as follows:

- **Simultaneous downloading of embedded images:** A Web browser typically establishes parallel connections to a server to retrieve multiple

embedded images at the same time. The first few bytes of an image typically indicate the size of the picture. This allows the browser to start rendering the Web page before the images have arrived in their entirety. The JPEG and GIF formats also support *progressive* encoding, or *interlacing,* which enables the browser to display the image with increasing quality as the response message arrives from the server. A user may prefer to see a coarse-grain view of several embedded images rather than a fine-grain view of a single image, making parallel downloading of embedded images an attractive alternative to serializing the requests. Parallel connections also arise when a user issues requests from multiple browser windows at the same time.

- **Proxy acting on behalf of multiple clients:** A Web proxy may handle requests for multiple clients accessing the same Web server at the same time. The proxy could conceivably send all of the requests over a single TCP connection. This avoids the overhead of establishing a new connection, at the expense of coupling the performance experienced by different users. Suppose user A and user B configure their browsers to connect to the same proxy. Suppose user B requests a 100-byte HTML file just after user A requested a 100 MB file from the same Web server. If the proxy sends both requests over the same (persistent) connection to the server, then user B would have to wait for the transmission of the 100-megabyte file to complete before any part of the small HTML file would be sent. Having two connections to the server would allow the proxy to retrieve the files in parallel, resulting in a much better response time for user B.
- **Higher throughput by transmitting aggressively:** The client can achieve higher throughput by establishing multiple TCP connections to the server. Suppose a client communicates with a server over a path with a large RTT. Even if the network and the server are lightly loaded, the high RTT limits the TCP throughput. The maximum window size is limited by the client's receive buffer and the congestion window, and the server cannot transmit more than one window's worth of data before receiving acknowledgments sent by the client. In addition, the congestion window is small at the beginning of the connection because of the slow-start phase of congestion control. Transmitting responses over multiple connections can increase the overall throughput from the server to the client.

The HTTP specification suggests that a user agent should have at most two TCP connections to a server at a time, and a proxy that handles requests on behalf of multiple clients should limit itself to two connections per requesting client. However, client and proxy implementations often disobey these guidelines. In addition, a user agent may open multiple parallel connections to retrieve different parts of a *single* resource by issuing range requests, as discussed in Chapter 7 (Section 7.4.1).

8.3.2 Problems with parallel connections

Parallel connections increase overall throughput for an individual client, at the expense of the greater good. The use of parallel connections causes several problems, as follows:

- **Unfairness to other clients:** A client that issues requests on multiple connections at a time achieves higher throughput by claiming bandwidth that would normally be allocated to other clients. Consider two Web clients that send requests over the same path through the network to the same Web server. Suppose that client A has four parallel TCP connections, whereas client B has just one TCP connection. Client A may receive up to four times more server processing and network bandwidth than client B, which could result in significantly lower latency for client A. However, client B would receive lower throughput and experience higher latency. One way to counteract the unfairness is for client B to open multiple TCP connections as well. To receive a higher proportion of the bandwidth, each client has an incentive to open more connections than the other clients.
- **Higher network and server load:** In addition to introducing unfairness between users, parallel connections increase the load on the server and the network. Consider a client that opens multiple connections at the same time to download a collection of embedded images. This introduces a sudden burst of traffic that imparts a heavy load on the network and the server. Even if each TCP connection proceeds through the slow-start phase of congestion control, the aggregate traffic from the set of connections could be quite large. As an extreme example, consider a client that opens 20 TCP connections to the same server. The client would generate 20 SYN packets in a very short period, followed shortly by 20 HTTP request messages. The Web server would consume most of its system resources trying to receive and process the SYN packets and HTTP requests. For the network, the collective load generated by a large number of SYN packets can cause a sudden backlog of traffic, ultimately leading to packet loss.
- **Higher user-perceived latency:** In addition to competing with traffic destined to other clients, the parallel connections compete with each other for network and server bandwidth. Consider a client with a 28.8 Kb/sec access link to the Internet (e.g., a telephone modem). If the client opens 20 parallel connections, each transfer receives at most 1.44 Kb/sec. In retrieving embedded images in a Web page, slow progress on multiple transfers may be preferable to fast progress on a single transfer. In other cases, the user may prefer fast progress on a small number of transfers. For example, a user may download multiple Web pages in different browser windows. Having a large number of active transfers at the same time would force the user to wait longer to receive any of the Web pages.

These fairness and performance problems have several possible remedies, as follows:

- **Removing performance incentives for parallel connections:** Enforcing fairness typically requires the use of scheduling algorithms to arbitrate access to network bandwidth and the system resources at the server. For example, rather than performing first-in first-out scheduling, a network router could alternate between packets to or from different end points. Alternatively, the router could keep track of how much traffic has been sent to each destination or from each source and penalize the end points that send too aggressively. The penalty may involve discarding some proportion of the packets. However, these techniques increase the complexity of the routers. Similarly, the TCP implementation on the Web server machine could ensure that traffic destined to different clients receives a fair share of the network bandwidth. In addition, the Web server could allocate its processing, memory, and disk resources fairly across different clients.
- **Providing alternatives to parallel connections:** Persistent connections address some of the incentives for using parallel connections, as discussed in Chapter 7 (Section 7.5). Sending multiple responses over a single TCP connection avoids the overhead of establishing multiple connections. In addition, reusing an existing TCP connection typically avoids the need to repeat the slow-start phase of congestion control. However, persistent connections do not address the desire to download multiple embedded images at the same time. A client could conceivably retrieve a portion of each image on a single persistent connection by sending a series of range requests. As an alternative, the relationship between HTTP and TCP could change to allow interleaving of several independent transfers on a single TCP connection.

Techniques for fair bandwidth allocation have been an active area of research and development as the Internet continues to evolve from a best-effort environment to a network that supports a wide range of services [Kes97]. Recent proposals for changing how end hosts allocate bandwidth across a collection of Web transfers are considered in more detail in Chapter 15 (Section 15.1).

8.4 Server Overheads

Reading request messages, generating responses, and transmitting data to clients consume significant resources at Web servers. These operations require the server to perform several functions that relate to TCP. In this section, we describe how some of these steps can be combined or avoided. Then we consider the challenges of handling a large number of simultaneous connections. These performance issues apply to both proxies and origin servers.

8.4.1 Combining system calls

The Web server interacts with TCP through a series of system calls. Performing multiple steps in a single system call provides the operating system with additional information that can improve the efficiency of the data transfer. Consider a UNIX-based Web server processing a `GET` request for a static file:

1. **Listening for requests for new connections:** The server listens for requests for new connections, and the operating system maintains an accept queue of pending connections.
2. **Establishing a new connection:** The server acquires a new connection from the accept queue using the *accept()* call. At this point, the server has a connection for communicating with the requesting client. In preparation for transmitting data to the client, the server may perform a *setsockopt()* call to disable Nagle's algorithm.
3. **Generating the response message:** The server performs one or more *read()* calls to read the client's HTTP request message. Then the server parses the request message and identifies the requested file. The server opens the requested file, using the *open()* call. The server may invoke various other system calls to construct the HTTP response header (e.g., to learn the current time, as well as the file's size and last modification time).
4. **Sending the response message:** After constructing the HTTP response header, the server uses the *write()* call to write these bytes to the connection. Then the server can begin sending the file. This involves using the *read()* call to read from the file and the *write()* call to write the data to the connection. If the send buffer is full, the operating system may prevent the server application from performing additional *write()* operations until some of the earlier data has been successfully transmitted to the client.
5. **Closing the file and (optionally) the connection:** After writing the last byte of the response, the server can *close()* the file and optionally *close()* the TCP connection. If the server implements persistent connections, and the client did not request that the connection be closed, the server may keep the connection open for a subsequent transfer. Finally, the server can perform a *write()* to create an entry in the server log.

Executing a system call typically requires a context switch between the server application and the operating system. Having a single system call perform multiple operations avoids the overhead and delay associated with context switches. Combining multiple steps into a single system call also facilitates other performance optimizations related to TCP [NBK99], as follows:

- **Copy the file contents directly to the operating system:** In satisfying the `GET` request, the Web server must read a file and then write the contents to the send buffer. In many cases, the server does not inspect or

manipulate the contents of the file. Reading the file, and then writing into the send buffer, introduces unnecessary overhead. The bytes are copied from the file system to user space and then from user space to the send buffer in the operating system. Instead, data could be copied directly from the file system to the send buffer using a single system call. Several operating systems include a *sendfile()* or *transmitfile()* system call to perform this function.

- **Send the response header and body with a single call:** The server could send the response header as part of the same system call, such as the *writev()* call. The server typically constructs the response header in a separate buffer. The combined system call would need to allow the application to specify a buffer (for the response header) and a file (for the response body) and then transmit the contents of the buffer followed by the contents of the file. This would avoid the need for separate system calls to write the header and the body to the socket, without requiring the application to first copy the header and body into a contiguous buffer. In addition, this enables the operating system to send the beginning of the response body in the same IP packet as the response header. Otherwise, if two system calls were involved, the operating system might send the header before the server performed a second call to send the body. Combining these two steps reduces the total number of packets and avoids potentially harmful interactions with the delayed-ACK timer, as discussed in Section 8.2.3.
- **Send the response and close the connection with a single call:** The server could close the connection as part of the same system call that writes the HTTP response. This would allow the server to write the HTTP header, send the response body, and close the connection in a single step. The operating system knows that the server wants to close the underlying TCP connection. Closing the connection involves sending a packet with the FIN bit set. The operating system has the option of piggybacking the FIN on the last data packet of the transfer, as shown in Figure 8.10. This is possible because the operating system already knows that the application wants to close the connection. Otherwise, the operating system may have already transmitted the last data packet before learning that the application intended to close the connection. For large response messages, the operating system typically would not have completed the transmission of the entire response before the second system call. However, smaller response messages are likely to require a separate FIN packet. Having a single system call for sending the response and closing the connection ensures that the operating system piggybacks the FIN. This reduces overhead by avoiding the transmission of the extra packet.

Extending the operating system to include new system calls can improve the performance of Web servers. In theory, the entire Web server could be

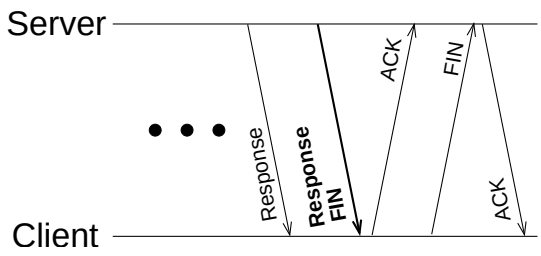

Figure 8.10. Server piggybacking the FIN on the last data packet

implemented in the operating system. Implementing an application in the operating system is common in small embedded systems that perform a single task. However, general-purpose operating systems usually do not implement entire applications. Executing time-consuming functions inside the operating system makes it difficult to share processor, memory, and disk resources across multiple tasks. As a result, operating systems typically have a small set of carefully chosen system calls. The needs of Web servers may motivate the addition of a few system calls or the improvement of the implementation of other system calls. However, individual applications usually do not warrant major changes in the set of functions provided by the operating system.

8.4.2 Managing multiple connections

The efficiency of a Web server depends on the number of TCP connections that are open simultaneously. Web servers control the number of connections in two main ways: by rejecting requests for new connections and by closing idle connections.

Controlling number of simultaneous connections

Each TCP connection consumes a certain amount of memory at the server for storing TCP state information and the send/receive buffer. For example, each TCP connection has a control block that stores information such as the congestion window and RTT estimates. These memory requirements can grow quite high when the server has a large number of simultaneous TCP connections. Inactive connections in the TIME_WAIT state also consume server resources, as discussed in Section 8.1.3. In addition, a process-driven server has a separate process for each open connection, resulting in additional memory requirements. TCP and process state consume memory that could be used for other purposes, such as server-side caching of Web resources and their metadata, as discussed in Chapter 4 (Section 4.3). Server-side caching becomes much less effective when the server has a very large number of simultaneous connections. Cache misses

introduce extra server overhead for fetching resources from disk and regenerating metadata.

The overhead for the operating system to handle an incoming packet grows with the number of open connections. When a packet arrives, the operating system must inspect the source and destination IP addressees and port numbers to demultiplex to the appropriate TCP connection. In some older operating systems, packet demultiplexing required a linear scan of the list of connections; as an optimization, the connections in the TIME_WAIT state could be placed at the end of the list because they are very unlikely to receive any packets. For a further reduction in latency, modern operating systems maintain a hash table that maps the source and destination IP addresses and port numbers to the appropriate connection. Still, the access latency and memory requirements for the hash table grow in the number of connections. In addition, for an event-driven server, a single process must listen (e.g., using the *select()* call) on all open connections for arriving packets. The overhead of this system call increases with the number of simultaneous connections, even when many of the connections are idle [BM98].

The optimal number of active TCP connections depends on the server architecture and the characteristics of the HTTP traffic. Generating and transmitting response messages consumes processor, memory, disk, and network resources at the server. Sharing these system resources across a large number of simultaneous requests results in higher delay in generating and transmitting the response messages. As an extreme example, consider a Web server that invokes a CGI script that requires two seconds of processing time to generate each response message. Allowing the processor to alternate back and forth between servicing each of the ten requests would result in a 20-second latency in generating each response. Higher user-perceived latency may cause the user to leave the Web site or abort the slow responses and initiate new requests. The poor performance irritates the users and wastes system resources at the server. In the worst case, the server becomes so overloaded that no productive work is performed. The number of simultaneous TCP connections must be carefully controlled to avoid this situation.

Despite the disadvantages of having too many simultaneous TCP connections, the server should not be unnecessarily restrictive. Limiting the number of simultaneous connections may block or delay TCP connection setup requests from new clients. When the server has reached the maximum number of connections, any arriving SYN packets are placed in a queue. The SYN requests are delayed until one of the existing connections closes. Once the accept queue is full, the server drops SYN packets, thereby rejecting requests for new connections. An overly conservative limit on the number of connections also results in underutilization of the server and network resources. The server needs a certain number of active TCP connections to exploit the available network bandwidth because the transmission rate for many connections is limited by round-trip

delays and throughput limitations along the path to the requesting client. A Web server could send responses to a large number of low-bandwidth clients without exhausting the capacity of a high-bandwidth link to the Internet.

Policies for closing persistent connections

As part of controlling the number of connections, an HTTP/1.1 server must also decide when to close a persistent connection. Keeping a persistent connection open allows the server to respond more quickly to subsequent requests by the same client and avoids the overheads of terminating and reestablishing the connection. Maintaining a connection beyond a single HTTP transfer increases the number of simultaneous connections at the server. The server can employ a wide variety of strategies for closing persistent connections. The simplest strategy is for the server to apply a timeout to close an idle connection. For example, the connection could be closed if no request has arrived in the last 15 seconds. If the connection has been idle for several seconds, the likelihood of receiving another request in the near future is very small.

However, even a relatively small timeout value may keep the connection open too long, especially because many clients request only one resource from the server. To handle this situation, the server can employ a hybrid timeout policy [PM95]. The server could use a small timeout after the first request and increase the timeout if the client issues additional requests. In addition to applying a timeout, the server may also limit the total number of requests allowed on a single connection. Otherwise, a client would have an incentive to generate periodic requests simply to keep the server from closing the persistent connection. For example, every ten seconds the browser might generate an HTTP request, even if the client is idle, in order to ensure that the connection remains open long enough to handle the client's next request. By limiting the number of requests per connection, the server avoids rewarding this behavior. However, this policy forces a well-behaved client to incur the overhead of establishing a new TCP connection after reaching the limit on the number of requests.

Policies based on timeouts and the number of requests are relatively easy to implement because they consider each TCP connection in isolation. In a process-driven server, these policies have the advantage of not requiring any coordination between processes. The server could conceivably implement more complicated policies by considering all of the open connections collectively. The server effectively has a cache of open TCP connections, and deciding which TCP connection to close amounts to a cache-replacement decision. For example, the server could close the connection that has been idle the longest. Similar to the timeout-based policy, this heuristic assumes that an idle connection is less likely to receive a request in the near future. Alternatively, the server could base the decision on the relative importance of the user. This may be a reasonable policy for a commercial site that wants to deliver good performance to high-paying

customers. Similarly, the server may try to avoid closing connections to far-away clients that would experience high setup delay in opening a new TCP connection.

8.5 Summary

Despite the benefits of dividing communication tasks into multiple protocol layers, interactions between layers can have negative implications on end-to-end application performance. The timers that control key operations in TCP have a direct effect on HTTP performance. TCP features that were designed when Telnet and Rlogin were dominant applications can interact in subtle ways with Web transfers. Implementation decisions in Web software components can mitigate or exacerbate these effects. In contrast to earlier Internet applications, a Web client typically has multiple transport-layer connections at the same time to download multiple Web resources in parallel. Parallel connections increase network and server load and introduce unfairness across Web users. Busy proxies and servers handle a large number of TCP connections on behalf of multiple clients. The efficiency of proxies and servers depends on having effective policies for controlling the number of open connections.