

## 12 TCP TRANSPORT

The standard transport protocols riding above the IP layer are **TCP** and **UDP**. As we saw in [11 UDP Transport](#), UDP provides simple datagram delivery to remote sockets, that is, to  $\langle \text{host}, \text{port} \rangle$  pairs. TCP provides a much richer functionality for sending data to (connected) sockets.

TCP is quite different in several dimensions from UDP. TCP is **stream-oriented**, meaning that the application can write data in very small or very large amounts and the TCP layer will take care of appropriate packetization (and also that TCP transmits a stream of bytes, not messages or records; *cf* [12.22.2 SCTP](#)). TCP is **connection-oriented**, meaning that a connection must be established before the beginning of any data transfer. TCP is **reliable**, in that TCP uses sequence numbers to ensure the correct order of delivery and a timeout/retransmission mechanism to make sure no data is lost short of massive network failure. Finally, TCP automatically uses the **sliding windows** algorithm to achieve throughput relatively close to the maximum available.

These features mean that TCP is very well suited for the transfer of large files. The two endpoints open a connection, the file data is written by one end into the connection and read by the other end, and the features above ensure that the file will be received correctly. TCP also works quite well for interactive applications where each side is sending and receiving streams of small packets. Examples of this include ssh or telnet, where packets are exchanged on each keystroke, and database connections that may carry many queries per second. TCP even works *reasonably* well for **request/reply** protocols, where one side sends a single message, the other side responds, and the connection is closed. The drawback here, however, is the overhead of setting up a new connection for each request; a better application-protocol design might be to allow multiple request/reply pairs over a single TCP connection.

Note that the connection-orientation and reliability of TCP represent abstract features built on top of the IP layer, which supports neither of them.

The connection-oriented nature of TCP warrants further explanation. With UDP, if a server opens a socket (the OS object, with corresponding socket address), then any client on the Internet can send to that socket, via its socket address. Any UDP application, therefore, must be prepared to check the source address of each packet that arrives. With TCP, all data arriving at a *connected* socket must come from the other endpoint of the connection. When a server S initially opens a socket s, that socket is “unconnected”; it is said to be in the LISTEN state. While it still has a socket address consisting of its host and port, a LISTENing socket will never receive data directly. If a client C somewhere on the Internet wishes to send data to s, it must first establish a connection, which will be defined by the **socketpair** consisting of the socket addresses (that is, the  $\langle \text{IP\_addr}, \text{port} \rangle$  pairs) at both C and S. As part of this connection process, a new *connected* child socket  $s_C$  will be created; it is  $s_C$  that will receive any data sent from C. Usually, the server will also create a new thread or process to handle communication with  $s_C$ . Typically the server will have multiple connected children of the original socket s, and, for each one, a process attached to it.

If C1 and C2 both connect to s, two connected sockets at S will be created,  $s_1$  and  $s_2$ , and likely two separate processes. When a packet arrives at S addressed to the socket address of s, the *source* socket address will also be examined to determine whether the data is part of the C1–S or the C2–S connection, and thus whether a read on  $s_1$  or on  $s_2$ , respectively, will see the data.

If S is acting as an ssh server, the LISTENing socket listens on port 22, and the connected child sockets correspond to the separate user login connections; the process on each child socket represents the login

process of that user, and may run for hours or days.

In Chapter 1 we likened TCP sockets to telephone connections, with the server like one high-volume phone number 800-BUY-NOWW. The unconnected socket corresponds to the number everyone dials; the connected sockets correspond to the actual calls. (This analogy breaks down, however, if one looks closely at the way such multi-operator phone lines are actually configured: each typically *does* have its own number.)

## 12.1 The End-to-End Principle

The End-to-End Principle is spelled out in [\[SRC84\]](#); it states in effect that transport issues are the responsibility of the endpoints in question and thus should not be delegated to the core network. This idea has been very influential in TCP design.

Two issues falling under this category are data corruption and congestion. For the first, even though essentially all links on the Internet have link-layer checksums to protect against data corruption, TCP still adds its own checksum (in part because of a history of data errors introduced *within* routers). For the latter, TCP is today essentially the *only* layer that addresses congestion management.

Saltzer, Reed and Clark categorized functions that were subject to the End-to-End principle this way:

The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)

This does not mean that the backbone Internet should not concern itself with congestion; it means that backbone congestion-management mechanisms should not completely replace end-to-end congestion management.

## 12.2 TCP Header

Below is a diagram of the TCP header. As with UDP, source and destination ports are 16 bits. The 4-bit Data Offset field specifies the number of 32-bit words in the header; if no options are present its value is 5.

As with UDP, the checksum covers the TCP header, the TCP data and an IP “pseudo header” that includes the source and destination IP addresses. The checksum must be updated by a NAT router that modifies any header values. (Although the IP and TCP layers are theoretically separate, and [RFC 793](#) in some places appears to suggest that TCP can be run over a non-IP internetwork layer, [RFC 793](#) also explicitly defines 4-byte addresses for the pseudo header. [RFC 2460](#) officially redefined the pseudo header to allow IPv6 addresses.)

		0	8	16	24	32					
		Source Port			Destination Port						
		Sequence Number									
		Acknowledgment Number									
Data Offset	Reserved	C W R	E C E	U R G	A C K	P S H	R S T	S Y N	F I N	Window Size	
		Checksum				Urgent Pointer					
		Options				Padding					

The **sequence** and **acknowledgment** numbers are for numbering the data, at the byte level. This allows TCP to send 1024-byte blocks of data, incrementing the sequence number by 1024 between successive packets, or to send 1-byte telnet packets, incrementing the sequence number by 1 each time. There is no distinction between DATA and ACK packets; all packets carrying data from A to B also carry the most current acknowledgment of data sent from B to A. Many TCP applications are largely unidirectional, in which case the sender would include essentially the same acknowledgment number in each packet while the receiver would include essentially the same sequence number.

It is traditional to refer to the data portion of TCP packets as **segments**.

### TCP History

The clear-cut division between the IP and TCP headers did not spring forth fully formed. See [\[CK74\]](#) for a discussion of a proto-TCP in which the sequence number (but not the acknowledgment number) appeared in the equivalent of the IP header (perhaps so it could be used for fragment reassembly).

The value of the sequence number, in *relative* terms, is the position of the first byte of the packet in the data stream, or the position of what would be the first byte in the case that no data was sent. The value of the acknowledgment number, again in relative terms, represents the byte position for the next byte expected. Thus, if a packet contains 1024 bytes of data and the first byte is number 1, then that would be the sequence number. The data bytes would be positions 1-1024, and the ACK returned would have acknowledgment number 1025.

The sequence and acknowledgment numbers, as sent, represent these relative values *plus* an **Initial Sequence Number**, or ISN, that is fixed for the lifetime of the connection. Each direction of a connection has its own ISN; see below.

TCP acknowledgments are **cumulative**: when an endpoint sends a packet with an acknowledgment number of N, it is acknowledging receipt of all data bytes numbered less than N. Standard TCP provides no mechanism for acknowledging receipt of packets 1, 2, 3 and 5; the highest cumulative acknowledgment that could be sent in that situation would be to acknowledge packet 3.

The TCP header defines some important flag bits; the brief definitions here are expanded upon in the sequel:

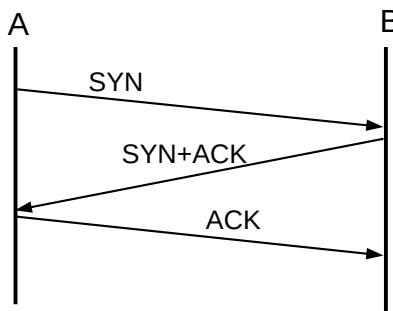
- **SYN**: for SYNchronize; marks packets that are part of the new-connection handshake

- **ACK**: indicates that the header Acknowledgment field is valid; that is, all but the first packet
- **FIN**: for FINish; marks packets involved in the connection closing
- **PSH**: for PuSH; marks “non-full” packets that should be delivered promptly at the far end
- **RST**: for ReSeT; indicates various error conditions
- **URG**: for URGent; part of a now-seldom-used mechanism for high-priority data
- **CWR** and **ECE**: part of the Explicit Congestion Notification mechanism, [14.8.3 Explicit Congestion Notification \(ECN\)](#)

## 12.3 TCP Connection Establishment

TCP connections are established via an exchange known as the **three-way handshake**. If A is the client and B is the LISTENing server, then the handshake proceeds as follows:

- A sends B a packet with the SYN bit set (a SYN packet)
- B responds with a SYN packet of its own; the ACK bit is now also set
- A responds to B’s SYN with its own ACK



TCP three-way handshake

Normally, the three-way handshake is triggered by an application’s request to connect; data can be sent only after the handshake completes. This means a one-RTT delay before any data can be sent. The original TCP standard [RFC 793](#) does allow data to be sent with the first SYN packet, as part of the handshake, but such data cannot be released to the remote-endpoint application until the handshake completes. Most traditional TCP programming interfaces offer no support for this early-data option.

There are recurrent calls for TCP to support data transmission within the handshake itself, so as to achieve request/reply turnaround comparable to that with RPC ([11.5 Remote Procedure Call \(RPC\)](#)). We return to this in [12.12 TCP Faster Opening](#).

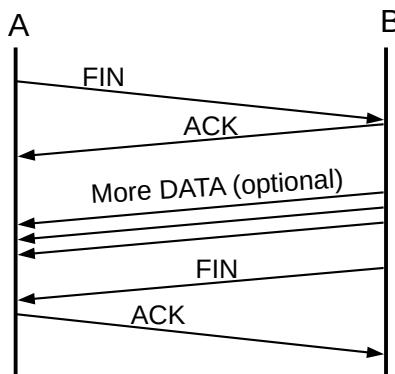
The three-way handshake is vulnerable to an attack known as **SYN flooding**. The attacker sends a large number of SYN packets to a server B. For each arriving SYN, B must allocate resources to keep track of what appears to be a legitimate connection request; with enough requests, B’s resources may face exhaustion. SYN flooding is easiest if the SYN packets are simply spoofed, with forged, untraceable source-IP addresses; see spoofing at [7.1 The IPv4 Header](#), and [12.10.1 ISNs and spoofing](#) below. SYN-flood attacks can also take the form of a large number of real connection attempts from a large number of real clients – often compromised and pressed into service by some earlier attack – but this requires considerably more resources

on the part of the attacker. See [12.22.2 SCTP](#) for an alternative handshake protocol (unfortunately not available to TCP) intended to mitigate SYN-flood attacks, at least from spoofed SYNs.

To **close** the connection, a superficially similar exchange involving FIN packets may occur:

- A sends B a packet with the FIN bit set (a FIN packet), announcing that it has finished sending data
- B sends A an ACK of the FIN
- B may continue to send additional data to A
- When B is also ready to cease sending, it sends its own FIN to A
- A sends B an ACK of the FIN; this is the final packet in the exchange

Here's the ladder diagram for this:



A typical TCP close

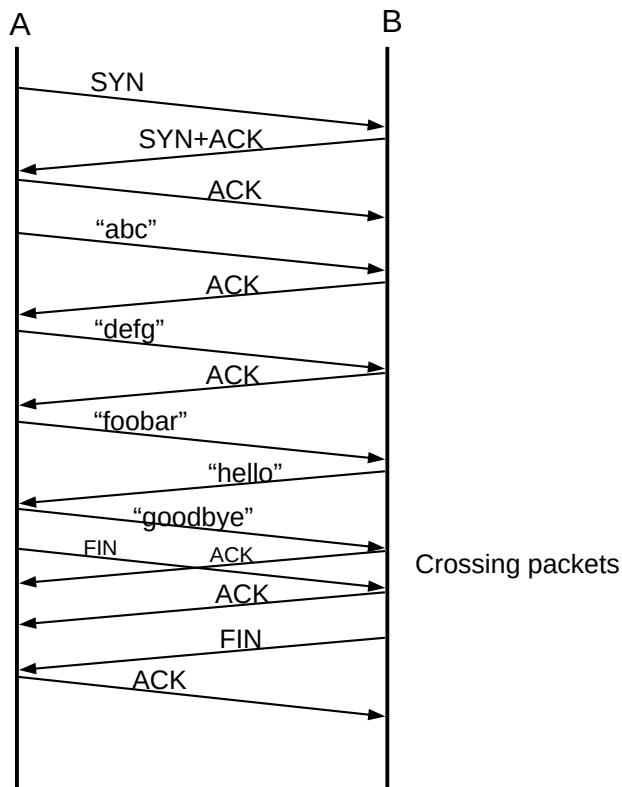
The FIN handshake is really more like two separate two-way FIN/ACK handshakes. We will return to TCP connection closing in [12.7.1 Closing a connection](#).

Now let us look at the full exchange of packets in a representative connection, in which A sends strings “abc”, “defg”, and “foobar” ([RFC 3092](#)). B replies with “hello”, and which point A sends “goodbye” and closes the connection. In the following table, *relative* sequence numbers are used, which is to say that sequence numbers begin with 0 on each side. The SEQ numbers in **bold** on the A side correspond to the ACK numbers in **bold** on the B side; they both count data flowing from A to B.

	A sends	B sends
1	SYN, seq=0	
2		SYN+ACK, seq=0, ack=1 (expecting)
3	ACK, seq=1, ack=1 (ACK of SYN)	
4	“abc”, seq=1, ack=1	
5		ACK, seq=1, ack=4
6	“defg”, seq=4, ack=1	
7		seq=1, ack=8
8	“foobar”, seq=8, ack=1	
9		seq=1, ack=14, “hello”
10	seq=14, ack=6, “goodbye”	
11,12	seq=21, ack=6, FIN	seq=6, ack=21 ;; ACK of “goodbye”, crossing packets
13		seq=6, ack=22 ;; ACK of FIN
14		seq=6, ack=22, FIN
15	seq=22, ack=7 ;; ACK of FIN	

(We will see below that this table is slightly idealized, in that real sequence numbers do *not* start at 0.)

Here is the ladder diagram corresponding to this connection:



In terms of the sequence and acknowledgment numbers, SYNs count as 1 byte, as do FINs. Thus, the SYN counts as sequence number 0, and the first byte of data (the “a” of “abc”) counts as sequence number 1. Similarly, the ack=21 sent by the B side is the acknowledgment of “goodbye”, while the ack=22 is the

acknowledgment of A's subsequent FIN.

Whenever B sends ACN=n, A follows by sending more data with SEQ=n.

TCP does *not* in fact transport relative sequence numbers, that is, sequence numbers as transmitted do not begin at 0. Instead, each side chooses its **Initial Sequence Number**, or **ISN**, and sends that in its initial SYN. The third ACK of the three-way handshake is an acknowledgment that the server side's SYN response was received correctly. All further sequence numbers sent are the ISN chosen by that side plus the relative sequence number (that is, the sequence number as if numbering did begin at 0). If A chose  $ISN_A=1000$ , we would add 1000 to all the bold entries above: A would send SYN(seq=1000), B would reply with  $ISN_B$  and ack=1001, and the last two lines would involve ack=1022 and seq=1022 respectively. Similarly, if B chose  $ISN_B=7000$ , then we would add 7000 to all the seq values in the "B sends" column and all the ack values in the "A sends" column. The table above up to the point B sends "goodbye", with actual sequence numbers instead of relative sequence numbers, is below:

	A, ISN=1000	B, ISN=7000
1	<b>SYN, seq=1000</b>	
2		<b>SYN+ACK, seq=7000, ack=1001</b>
3	<b>ACK, seq=1001, ack=7001</b>	
4	"abc", <b>seq=1001, ack=7001</b>	
5		<b>ACK, seq=7001, ack=1004</b>
6	"defg", <b>seq=1004, ack=7001</b>	
7		<b>seq=7001, ack=1008</b>
8	"foobar", <b>seq=1008, ack=7001</b>	
9		<b>seq=7001, ack=1014, "hello"</b>
10	<b>seq=1014, ack=7006, "goodbye"</b>	

If B had not been LISTENing at the port to which A sent its SYN, its response would have been **RST** ("reset"), meaning in this context "connection refused". Similarly, if A sent data to B before the SYN packet, the response would have been RST.

Finally, a RST can be sent by either side at any time to abort the connection. Sometimes routers along the path send "spoofed" RSTs to tear down TCP connections they are configured to regard as undesired; see [7.7.2 Middleboxes](#) and [RFC 3360](#). Worse, sometimes external attackers are able to tear down a TCP connection with a spoofed RST; this requires brute-force guessing the endpoint port numbers and the current SYN value ([RFC 793](#) does not require the RST packet's ACK value to match). In the days of 4 kB window sizes, guessing a valid SYN was a one-in-a-million chance, but window sizes have steadily increased ([14.9 The High-Bandwidth TCP Problem](#)); a 4 MB window size makes SYN guessing quite feasible. See also [RFC 4953](#), the RST-validation fix proposed in [RFC 5961](#) §3.2, and exercise 6.5.

If A sends a series of small packets to B, then B has the option of assembling them into a full-sized I/O buffer before releasing them to the receiving application. However, if A sets the **PSH** bit on each packet, then B should release each packet immediately to the receiving application. In Berkeley Unix and most (if not all) BSD-derived socket-library implementations, there is in fact no way to set the PSH bit; it is set automatically for each write. (But even this is not *guaranteed* as the sender may leave the bit off or consolidate several PuSHed writes into one packet; this makes using the PSH bit as a record separator difficult. In the program written to generate the Wireshark packet trace, below, most of the time the strings "abc", "defg", etc were PuSHed separately but occasionally they were consolidated into one packet.)

As for the **URG** bit, imagine a telnet (or ssh) connection, in which A has sent a large amount of data to B, which is momentarily stalled processing it. The application at A wishes to abort that processing by sending the interrupt character CNTL-C. Under normal conditions, the application at B would have to finish processing all the pending data before getting to the CNTL-C; however, the use of the URG bit can enable immediate asynchronous delivery of the CNTL-C. The bit is set, and the TCP header's Urgent Pointer field points to the CNTL-C in the current packet, far ahead in the normal data stream. The receiving application then skips ahead in its processing of the arriving data stream until it reaches the urgent data. For this to work, the receiving application process must have signed up to receive an asynchronous signal when urgent data arrives.

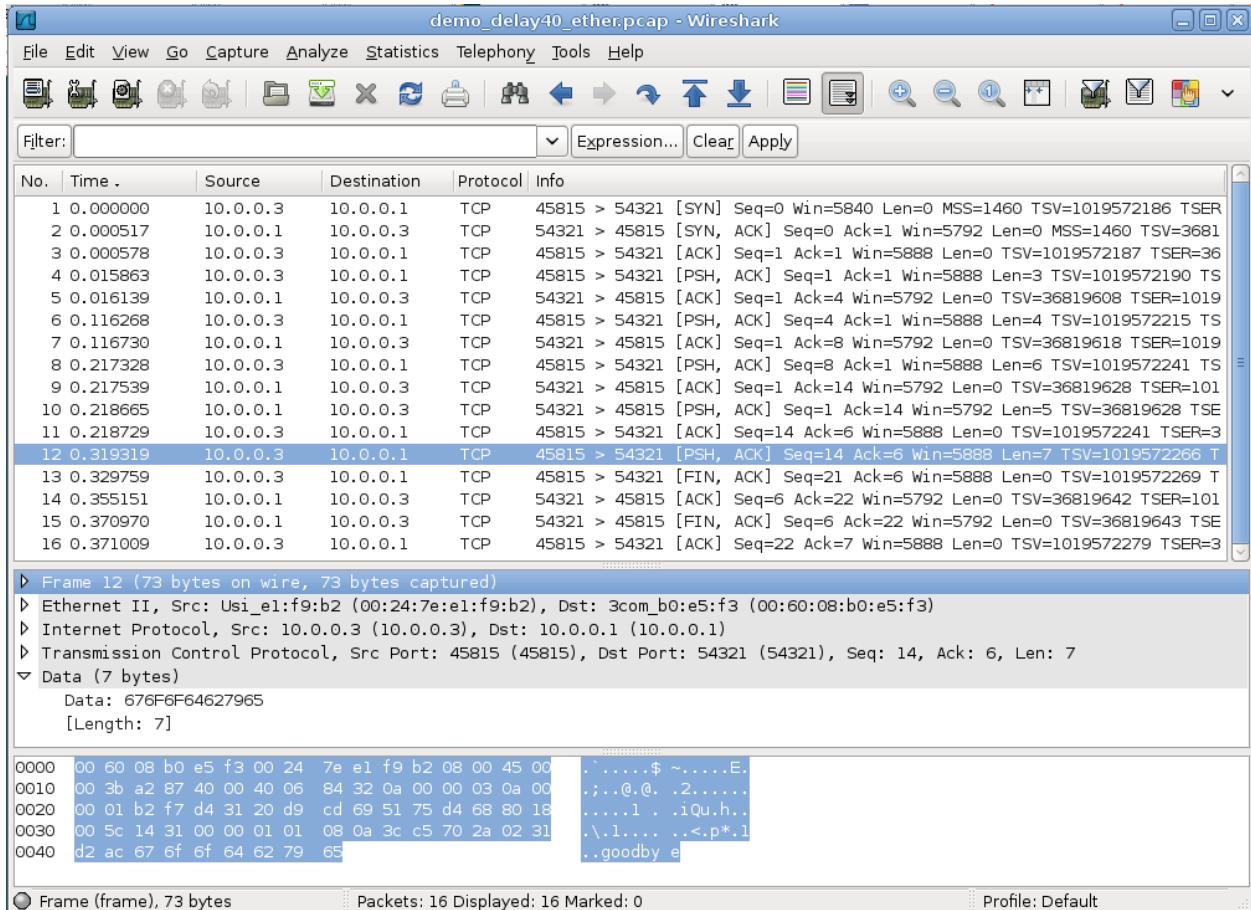
The urgent data does appear as part of the ordinary TCP data stream, and it is up to the protocol to determine the start of the data that is to be considered urgent, and what to do with the unread, buffered data sent ahead of the urgent data. For the CNTL-C example in the telnet protocol ([RFC 854](#)), the urgent data might consist of the telnet “Interrupt Process” byte, preceded by the “Interpret as Command” escape byte, and the earlier data is simply discarded.

Officially, the Urgent Pointer value, when the **URG** bit is set, contains the offset from the start of the current packet data to the *end* of the urgent data; it is meant to tell the receiver “you should read up to this point as soon as you can”. The original intent was for the urgent pointer to mark the last byte of the urgent data, but §3.1 of [RFC 793](#) got this wrong and declared that it pointed to the first byte *following* the urgent data. This was corrected in [RFC 1122](#), but most implementations to this day abide by the “incorrect” interpretation. [RFC 6093](#) discusses this and proposes, first, that the near-universal “incorrect” interpretation be accepted as standard, and, second, that developers avoid the use of the TCP urgent-data feature.

## 12.4 TCP and Wireshark

Below is a screenshot of the [Wireshark](#) program displaying a tcpdump capture intended to represent the TCP exchange above. Both hosts involved in the packet exchange were Linux systems. Side A uses socket address  $\langle 10.0.0.3, 45815 \rangle$  and side B (the server) uses  $\langle 10.0.0.1, 54321 \rangle$ .

WireShark is displaying *relative* TCP sequence numbers. The first three packets correspond to the three-way handshake, and packet 4 is the first data packet. Every data packet has the flags [PSH, ACK] displayed. The data in the packet can be inferred from the WireShark Len field, as each of the data strings sent has a different length.



The packets are numbered the same as in the table above up through packet 8, containing the string “foobar”. At that point the table shows B replying by a combined ACK plus the string “hello”; in fact, TCP sent the ACK alone and then the string “hello”; these are WireShark packets 9 and 10 (note packet 10 has Len=5). Wireshark packet 11 is then a standalone ACK from A to B, acknowledging the “hello”. Wireshark packet 12 (the packet highlighted) then corresponds to table packet 10, and contains “goodbye” (Len=7); this string can be seen at the right side of the bottom pane.

The table view shows A’s FIN (packet 11) crossing with B’s ACK of “goodbye” (packet 12). In the Wireshark view, A’s FIN is packet 13, and is sent about 0.01 seconds after “goodbye”; B then ACKs them both with packet 14. That is, the table-view packet 12 does not exist in the Wireshark view.

Packets 11, 13, 14 and 15 in the table and 13, 14, 15 and 16 in the Wireshark screen dump correspond to the connection closing. The program that generated the exchange at B’s side had to include a “sleep” delay of 40 ms between detecting the closed connection (that is, reading A’s FIN) and closing its own connection (and sending its own FIN); otherwise the ACK of A’s FIN traveled in the same packet with B’s FIN.

The ISN for A in this example was 551144795 and B’s ISN was 1366676578. The actual pcap packet-capture file is at [demo\\_tcp\\_connection.pcap](#).

## 12.5 TCP Offloading

In the Wireshark example above, the hardware involved used **TCP checksum offloading**, or TCO, to have the network-interface card do the actual checksum calculations; this permits a modest amount of parallelism. As a result, the checksums for outbound packets are wrong in the capture file. Wireshark has an option to disable the reporting of this.

It is also possible, with many newer network-interface cards, to offload the TCP segmentation process to the LAN hardware; this is most useful when the application is writing data continuously and is known as **TCP segmentation offloading**, or TSO. The use of TSO requires TCO, but not vice-versa.

TSO can be divided into large send offloading, LSO, for outbound traffic, and large receive offloading, LRO, for inbound. For outbound offloading, the host system transfers to the network card a large buffer of data (perhaps 64 kB), together with information about the headers. The network card then divides the buffer into 1500-byte packets, with proper TCP/IP headers, and sends them off.

For inbound offloading, the network card accumulates multiple inbound packets that are part of the same TCP connection, and consolidates them in proper sequence to one much larger packet. This means that the network card, upon receiving one packet, must wait to see if there will be more. This wait is very short, however, at most a few milliseconds. Specifically, all consolidated incoming packets must have the same TCP Timestamp value ([12.11 Anomalous TCP scenarios](#)).

TSO is of particular importance at very high bandwidths. At 10 Gbps, a system can send or receive close to a million packets per second, and offloading some of the packet processing to the network card can be essential to maintaining high throughput. TSO allows a host system to behave as if it were reading or writing very large packets, and yet the actual packet size on the wire remains at the standard 1500 bytes.

On Linux systems, the status of TCO and TSO can be checked using the command `ethtool --show-offload interface`. TSO can be disabled with `ethtool --offload interface tso off`.

## 12.6 TCP simplex-talk

Here is a Java version of the simplex-talk server for TCP. As with the UDP version, we start by setting up the socket, here a `ServerSocket` called `ss`. This socket remains in the LISTEN state throughout. The main while loop then begins with the call `ss.accept()` at the start; this call blocks until an incoming connection is established, at which point it returns the connected child socket `s`. The `accept()` call models the TCP protocol behavior of waiting for three-way handshakes initiated by remote hosts and, for each, setting up a new connection.

Connections will be accepted from *all* IP addresses of the server host, *eg* the “normal” IP address, the loopback address 127.0.0.1 and, if the server is multihomed, any additional IP addresses. Unlike the UDP case ([11.1.3.2 UDP and IP addresses](#)), **RFC 1122** requires (§4.2.3.7) that server response packets always be sent from the same server IP address that the client first used to contact the server. (See exercise 13.0 for an example of non-compliance.)

A server application can process these connected children either serially or in parallel. The stalk version here can handle both situations, either one connection at a time (`THREADING = false`), or by creating a new thread for each connection (`THREADING = true`). Either way, the connected child socket is turned over to `line_talker()`, either as a synchronous procedure call or as a new thread. Data is then read from

the socket's associated `InputStream` using the ordinary `read()` call, versus the `receive()` used to read UDP packets. The main loop within `line_talker()` does not terminate until the client closes the connection (or there is an error).

In the serial, non-threading mode, if a second client connection is made while the first is still active, then data can be sent on the second connection but it sits in limbo until the first connection closes, at which point control returns to the `ss.accept()` call, the second connection is processed, and the second connection's data suddenly appears.

In the threading mode, the main loop spends almost all its time waiting in `ss.accept()`; when this returns a child connection we immediately spawn a new thread to handle it, allowing the parent process to go back to `ss.accept()`. This allows the program to accept multiple concurrent client connections, like the UDP version.

The code here serves as a very basic example of the creation of Java threads. The inner class `Talker` has a `run()` method, needed to implement the `Runnable` interface. To start a new thread, we create a new `Talker` instance; the `start()` call then begins `Talker.run()`, which runs for as long as the client keeps the connection open. The file here is `tcp_stalks.java`.

```
/* THREADED simplex-talk TCP server */
/* can handle multiple CONCURRENT client connections */
/* newline is to be included at client side */

import java.net.*;
import java.io.*;

public class tstalks {

    static public int destport = 5431;
    static public int bufsize = 512;
    static public boolean THREADING = true;

    static public void main(String args[]) {
        ServerSocket ss;
        Socket s;
        try {
            ss = new ServerSocket(destport);
        } catch (IOException ioe) {
            System.err.println("can't create server socket");
            return;
        }
        System.err.println("server starting on port " + ss.getLocalPort());

        while(true) { // accept loop
            try {
                s = ss.accept();
            } catch (IOException ioe) {
                System.err.println("Can't accept");
                break;
            }

            if (THREADING) {
                Talker talk = new Talker(s);

```

```

        (new Thread(talk)).start();
    } else {
        line_talker(s);
    }
} // accept loop
} // end of main

public static void line_talker(Socket s) {
    int port = s.getPort();
    InputStream istr;
    try { istr = s.getInputStream(); }
    catch (IOException ioe) {
        System.err.println("cannot get input stream");           // most likely cause: s was closed
        return;
    }
    System.err.println("New connection from <" +
        s.getInetAddress().getHostAddress() + "," + s.getPort() + ">");
    byte[] buf = new byte[bufsize];
    int len;

    while (true) {          // while not done reading the socket
        try {
            len = istr.read(buf, 0, bufsize);
        }
        catch (SocketTimeoutException ste) {
            System.out.println("socket timeout");
            continue;
        }
        catch (IOException ioe) {
            System.err.println("bad read");
            break;           // probably a socket ABORT; treat as a close
        }
        if (len == -1) break;           // other end closed gracefully
        String str = new String(buf, 0, len);
        System.out.print("") + port + ":" + str); // str should contain newline
    } //while reading from s

    try {istr.close();}
    catch (IOException ioe) {System.err.println("bad stream close");}
    return;
    try {s.close();}
    catch (IOException ioe) {System.err.println("bad socket close");}
    return;
    System.err.println("socket to port " + port + " closed");
} // line_talker

static class Talker implements Runnable {
    private Socket _s;

    public Talker (Socket s) {

```

```

        _s = s;
    }

    public void run() {
        line_talker(_s);
    } // run
} // class Talker
}

```

### 12.6.1 The TCP Client

Here is the corresponding client `tcp_stalkc.java`. As with the UDP version, the default host to connect to is `localhost`. We first call `InetAddress.getByName()` to perform the DNS lookup. Part of the construction of the `Socket` object is the connection to the desired dest and destport. Within the main while loop, we use an ordinary `write()` call to write strings to the socket's associated `OutputStream`.

```

// TCP simplex-talk CLIENT in java

import java.net.*;
import java.io.*;

public class stalkc {

    static public BufferedReader bin;
    static public int destport = 5431;

    static public void main(String args[]) {
        String desthost = "localhost";
        if (args.length >= 1) desthost = args[0];
        bin = new BufferedReader(new InputStreamReader(System.in));

        InetAddress dest;
        System.err.print("Looking up address of " + desthost + "...");
        try {
            dest = InetAddress.getByName(desthost);
        }
        catch (UnknownHostException uhe) {
            System.err.println("unknown host: " + desthost);
            return;
        }
        System.err.println(" got it!");

        System.err.println("connecting to port " + destport);
        Socket s;
        try {
            s = new Socket(dest, destport);
        }
        catch(IOException ioe) {
            System.err.println("cannot connect to <" + desthost + ",", " +
destport + ">");
            return;
        }
    }
}

```

```
}

OutputStream sout;
try {
    sout = s.getOutputStream();
}
catch (IOException ioe) {
    System.err.println("I/O failure!");
    return;
}

//=====

while (true) {
    String buf;
    try {
        buf = bin.readLine();
    }
    catch (IOException ioe) {
        System.err.println("readLine() failed");
        return;
    }
    if (buf == null) break;      // user typed EOF character

    buf = buf + "\n";           // protocol requires sender includes \n
    byte[] bbuf = buf.getBytes();

    try {
        sout.write(bbuf);
    }
    catch (IOException ioe) {
        System.err.println("write() failed");
        return;
    }
} // while
}
```

A Python3 version of the stalk client is available at [tcp\\_stalkc.py](#).

Here are some things to try with THREADING=false in the server:

- start up two clients while the server is running. Type some message lines into both. Then exit the first client.
- start up the client before the server.
- start up the server, and then the client. Kill the server and then type some message lines to the client. What happens to the client? (It may take a couple message lines.)
- start the server, then the client. Kill the server and restart it. Now what happens to the client?

With THREADING=true, try connecting multiple clients simultaneously to the server. How does this behave differently from the first example above?

See also exercise 14.0.

### 12.6.2 netcat again

As with UDP ([11.1.4 netcat](#)), we can use the `netcat` utility to act as either end of the TCP simplex-talk connection. As the client we can use

```
netcat localhost 5431
```

while as the server we can use

```
netcat -l -k 5431
```

Here (but not with UDP) the `-k` option causes the server to accept multiple connections in sequence. The connections are handled one at a time, as is the case in the stalk server above with `THREADING=false`.

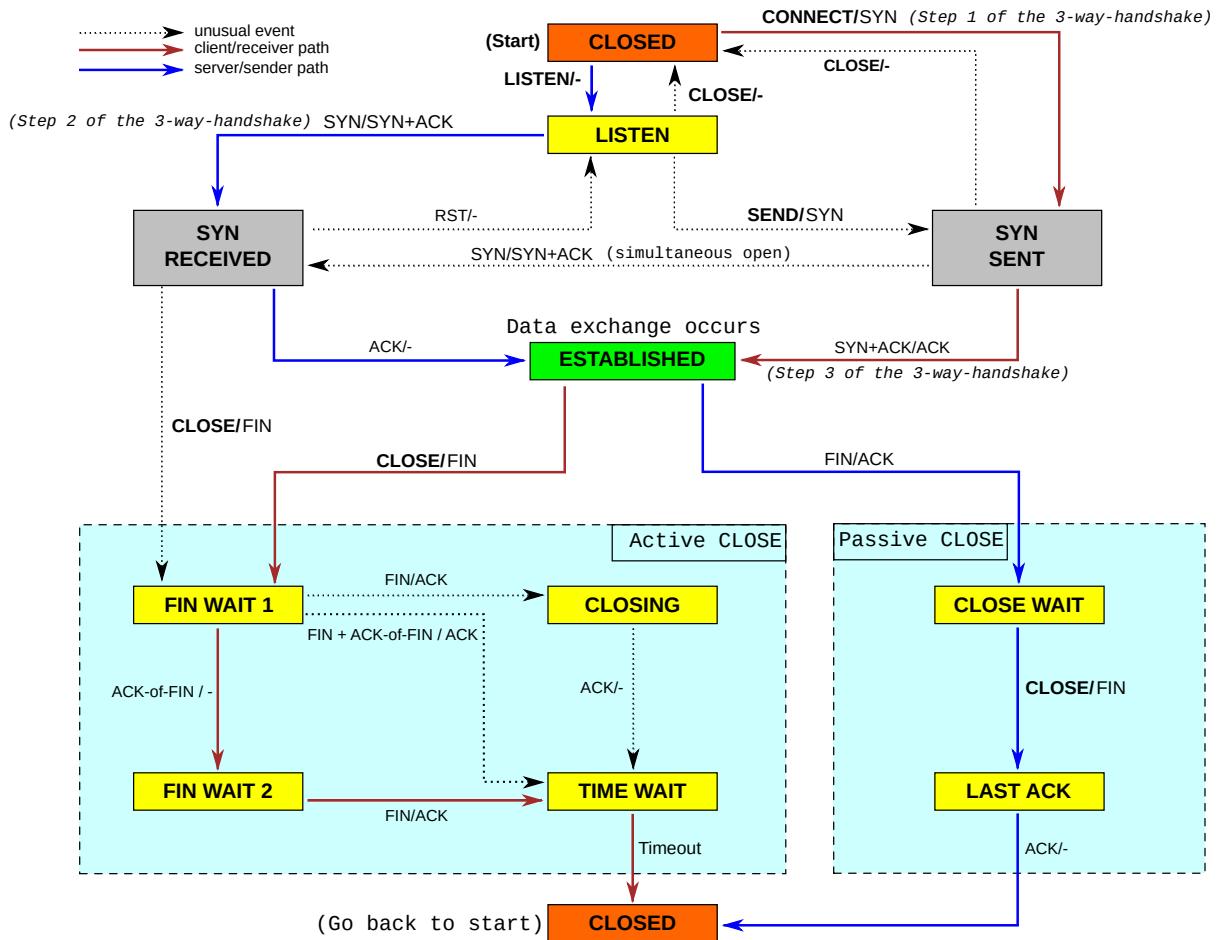
We can also use `netcat` to download web pages, using the [HTTP](#) protocol. The command below sends an HTTP GET request (version 1.1; [RFC 2616](#) and updates) to retrieve part of the website for this book; it has been broken over two lines for convenience.

```
echo -e 'GET /index.html HTTP/1.1\r\nHOST: intronetworks.cs.luc.edu\r\n' |  
netcat intronetworks.cs.luc.edu 80
```

The `\r\n` represents the officially mandatory carriage-return/newline line-ending sequence, though `\n` will often work. The `index.html` identifies the file being requested; as `index.html` is the default it is often omitted, though the preceding `/` is still required. The webserver may support other websites as well via virtual hosting ([7.8.1 nslookup \(and dig\)](#)); the `HOST:` specification identifies to the server the specific site we are looking for. Version 2 of HTTP is described in [RFC 7540](#); its primary format is binary. (For production command-line retrieval of web pages, [cURL](#) and [wget](#) are standard choices.)

## 12.7 TCP state diagram

A formal definition of TCP involves the **state diagram**, with conditions for transferring from one state to another, and responses to all packets from each state. The state diagram originally appeared in [RFC 793](#); the following diagram came from [http://commons.wikimedia.org/wiki/File:Tcp\\_state\\_diagram\\_fixed.svg](http://commons.wikimedia.org/wiki/File:Tcp_state_diagram_fixed.svg). The blue arrows indicate the sequence of state transitions typically followed by the server; the brown arrows represent the client. Arrows are labeled with **event / action**; that is, we move from LISTEN to SYN\_RECV upon receipt of a SYN packet; the action is to respond with SYN+ACK.

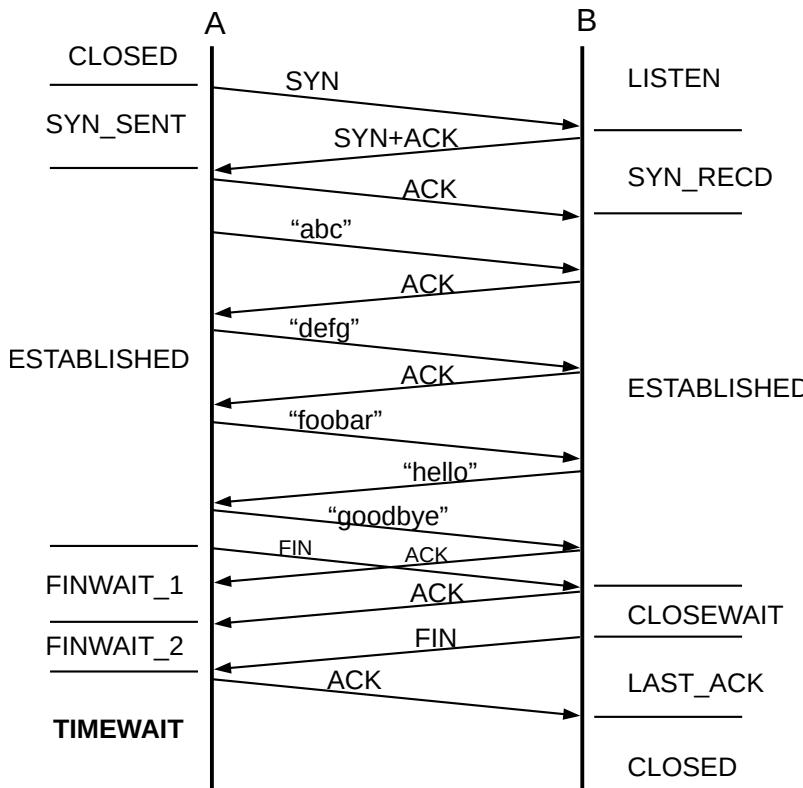


In general, this state-machine approach to protocol specification has proven very effective, and is used for most protocols. It makes it very clear to the implementer how the system should respond to each packet arrival. It is also a useful model for the implementation itself.

It is visually impractical to list every possible transition within the state diagram, full details are usually left to the accompanying text. For example, although this does not appear in the state diagram above, the per-state response rules of TCP require that in the ESTABLISHED state, if the receiver sends an ACK outside the current sliding window, then the correct response is to reply with one's own current ACK. This includes the case where the receiver *acknowledges data not yet sent*.

The ESTABLISHED state and the states below it are sometimes called the **synchronized** states, as in these states both sides have confirmed one another's ISN values.

Here is the ladder diagram for the 14-packet connection described above, this time labeled with TCP states.



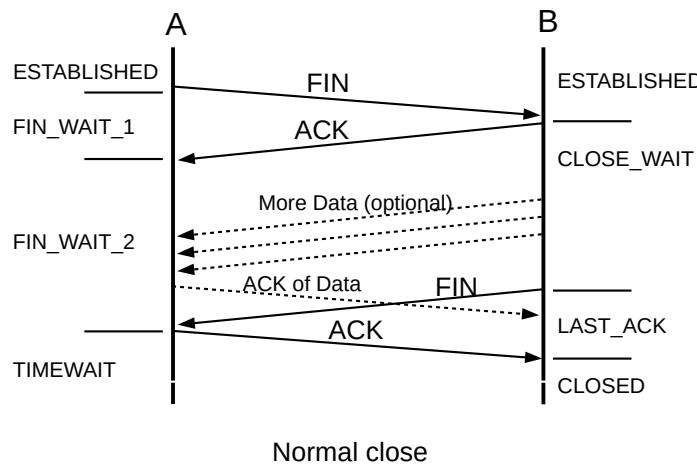
Although it essentially never occurs in practice, it is possible for each side to send the other a SYN, requesting a connection, **simultaneously** (that is, the SYNs cross on the wire). The telephony analogue occurs when each party dials the other simultaneously. On traditional land-lines, each party then gets a busy signal. On cell phones, your mileage may vary. With TCP, a single connection is created. With OSI TP4, two connections are created. The OSI approach is not possible in TCP, as a connection is determined only by the socketpair involved; if there is only one socketpair then there can be only one connection.

It is possible to view connection states under either Linux or Windows with `netstat -a`. Most states are ephemeral, exceptions being LISTEN, ESTABLISHED, TIMEWAIT, and CLOSE\_WAIT. One sometimes sees large numbers of connections in CLOSE\_WAIT, meaning that the remote endpoint has closed the connection and sent its FIN, but the process at your end has not executed `close()` on its socket. Often this represents a programming error; alternatively, the process at the local end is still working on something. Given a local port number p in state CLOSE\_WAIT on a Linux system, the (privileged) command `lsof -i :p` will identify the process using port p.

The reader who is implementing TCP is encouraged to consult [RFC 793](#) and updates. For the rest of us, below are a few general observations about closing connections.

### 12.7.1 Closing a connection

The “normal” TCP close sequence is as follows:



A's FIN is, in effect, a promise to B not to *send* any more. However, A must still be prepared to receive data from B, hence the optional data shown in the diagram. A good example of this occurs when A is sending a stream of data to B to be sorted; A sends FIN to indicate that it is done sending, and only then does B sort the data and begin sending it back to A. This can be generated with the command, on A, `cat thefile | ssh B sort`. That said, the presence of the optional B-to-A data above following A's FIN is relatively less common.

In the diagram above, A sends a FIN to B and receives an ACK, and then, later, B sends a FIN to A and receives an ACK. This essentially amounts to two separate two-way closure handshakes.

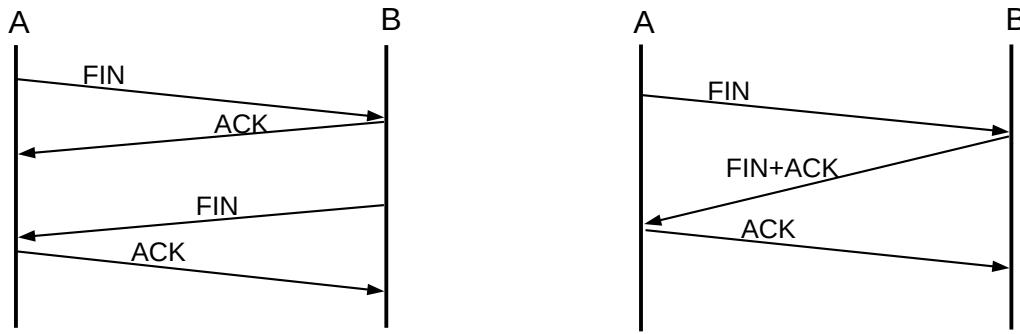
Either side may elect to close the connection, just as either party to a telephone call may elect to hang up. The first side to send a FIN – A in the diagram above – takes the **Active CLOSE** path; the other side takes the **Passive CLOSE** path. In the diagram, active-closer A moves from state ESTABLISHED to FIN\_WAIT\_1 to FIN\_WAIT\_2 (upon receipt of B's ACK of A's FIN), and then to TIMEWAIT and finally to CLOSED. Passive-closer B moves from ESTABLISHED to CLOSE\_WAIT to LAST\_ACK to CLOSED.

A **simultaneous close** – having both sides send each other FINs before receiving the other side's FIN – is a little more likely than a simultaneous open, earlier above, though still not very. Each side would send its FIN and move to state FIN\_WAIT\_1. Then, upon receiving each other's FIN packets, each side would send its final ACK and move to CLOSING. See exercises 4.0 and 4.5.

A TCP endpoint is **half-closed** if it has sent its FIN (thus promising not to send any more data) and is waiting for the other side's FIN; this corresponds to A in the diagram above in states FIN\_WAIT\_1 and FIN\_WAIT\_2. With the BSD socket library, an application can half-close its connection with the appropriate call to `shutdown()`.

Unrelatedly, A TCP endpoint is **half-open** if it is in the ESTABLISHED state, but during a lull in the exchange of packets the other side has rebooted; this has nothing to do with the close protocol. As soon as the ESTABLISHED side sends a packet, the rebooted side will respond with RST and the connection will be fully closed.

In the absence of the optional data from B to A after A sends its FIN, the closing sequence reduces to the left-hand diagram below:



Two TCP close scenarios with no B-to-A data

If B is ready to close immediately, it is possible for B’s ACK and FIN to be combined, as in the right-hand diagram above, in which case the resultant diagram superficially resembles the connection-opening three-way handshake. In this case, A moves directly from FIN\_WAIT\_1 to TIMEWAIT, following the state-diagram link labeled “FIN + ACK-of-FIN”. In theory this is rare, as the ACK of A’s FIN is generated by the kernel but B’s FIN cannot be sent until B’s process is scheduled to run on the CPU. If the TCP layer adopts a policy of *immediately* sending ACKs upon receipt of any packet, this will never happen, as the ACK will be sent well before B’s process can be scheduled to do anything. However, if B *delays* its ACKs slightly (and if it has no more data to send), then it is possible – and in fact not uncommon – for B’s ACK and FIN to be sent together. Delayed ACKs, are, as we shall see below, a common strategy ([12.15 TCP Delayed ACKs](#)). To create the scenario of [12.4 TCP and Wireshark](#), it was necessary to introduce an artificial delay to prevent the simultaneous transmission of B’s ACK and FIN.

### 12.7.2 Calling `close()`

Most network programming interfaces provide a `close()` method for ending a connection, based on the close operation for files. However, it usually closes bidirectionally and so models the TCP closure protocol rather imperfectly.

As we have seen in the previous section, the TCP close sequence is followed more naturally if the active-closing endpoint calls `shutdown()` – promising not to send more, but allowing for continued receiving – before the final `close()`. Here is what *should* happen at the application layer if endpoint A of a TCP connection wishes to initiate the closing of its connection with endpoint B:

- A’s application calls `shutdown()`, thereby promising not to send any more data. A’s FIN is sent to B. A’s application is expected to continue reading, however.
- The connection is now half-closed. On receipt of A’s FIN, B’s TCP layer knows this. If B’s application attempts to read more data, it will receive an end-of-file indication (this is typically a `read()` or `recv()` operation that returns immediately with 0 bytes received).
- B’s application is now done reading data, but it may or may not have more data to send. When B’s application is done sending, it calls `close()`, at which point B’s FIN is sent to A. Because the connection is already half-closed, B’s `close()` is really a second half-close, ending further transmission by B.
- A’s application keeps reading until it too receives an end-of-file indication, corresponding to B’s FIN.

- The connection is now fully closed. No data has been lost.

It is sometimes the case that it is evident to A from the application protocol that B will not send more data. In such cases, A might simply call `close()` instead of `shutdown()`. This is risky, however, unless the protocol is crystal clear: if A calls `close()` and B later does send a little more data after all, or if B has already sent some data but A has not actually read it, A's TCP layer may send RST to B to indicate that not all B's data was received properly. [RFC 1122](#) puts it this way:

If such a host issues a CLOSE call while received data is still pending in TCP, or if new data is received after CLOSE is called, its TCP SHOULD send a RST to show that data was lost.

If A's RST arrives at B before all of A's sent data has been processed by B's application, it is entirely possible that data sent by A will be lost, that is, will never be read by B.

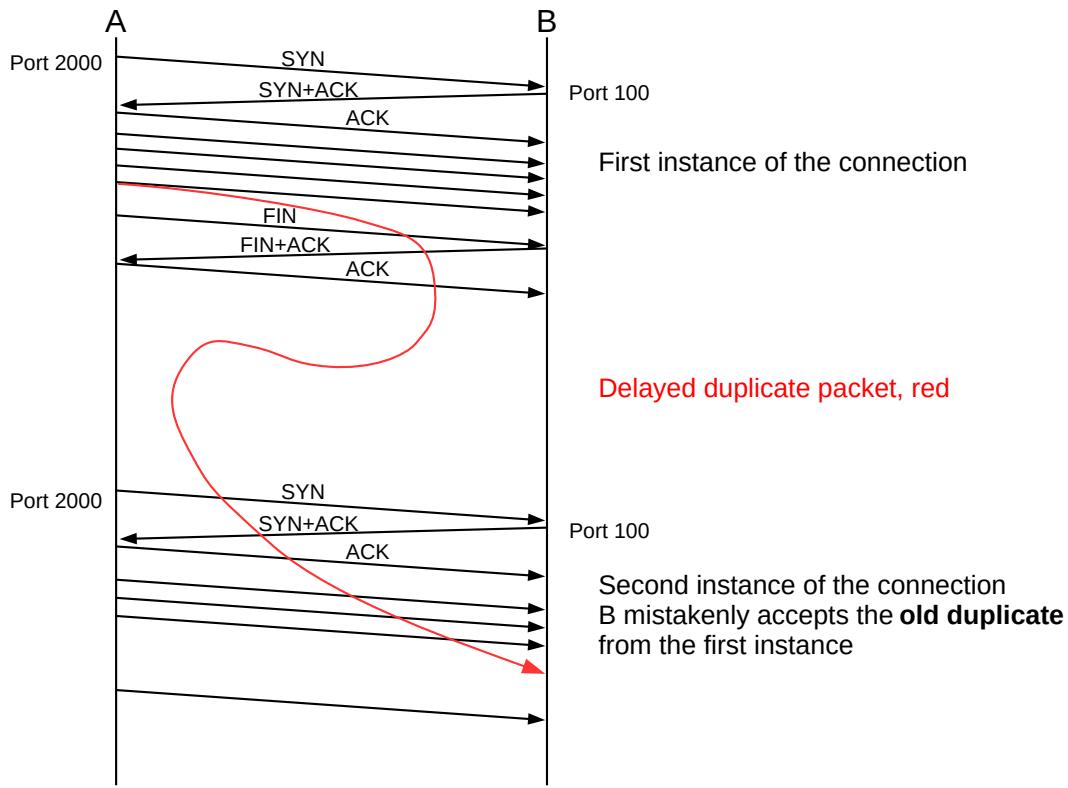
In the BSD socket library, A can set the `SO_LINGER` option, which causes A's `close()` to block until A's data has been delivered to B (or until the `SO_LINGER` timeout has expired). However, `SO_LINGER` has no bearing on the issue above; post-close data from B to A will still cause A to send a RST.

In the simplex-talk program at [12.6 TCP simplex-talk](#), the client does not call `shutdown()` (it implicitly calls `close()` when it exits). When the client is done, the server calls `s.close()`. However, the fact that there is no data at all sent from the server to the client prevents the problem discussed above.

See exercises 14.0 and 15.0.

## 12.8 TCP Old Duplicates

Conceptually, perhaps the most serious threat facing the integrity of TCP data is external old duplicates ([11.3 Fundamental Transport Issues](#)), that is, very late packets from a previous instance of the connection. Suppose a TCP connection is opened between A and B. One packet from A to B is duplicated and unduly delayed, with sequence number N. The connection is closed, and then another instance is reopened, that is, a connection is created using the same ports. At some point in the second connection, when an arriving packet with `seq=N` would be acceptable at B, the old duplicate shows up. Later, of course, B is likely to receive a `seq=N` packet from the new instance of the connection, but that packet will be seen by B as a duplicate (even though the data does not match), and (we will assume) be ignored.



For TCP, it is the actual sequence numbers, rather than the relative sequence numbers, that would have to match up. The diagram above ignores that.

As with TFTP, coming up with a possible scenario accounting for the generation of such a late packet is not easy. Nonetheless, many of the design details of TCP represent attempts to minimize this risk.

Solutions to the old-duplicates problem generally involve setting an upper bound on the lifetime of any packet, the MSL, as we shall see in the next section. T/TCP ([12.12 TCP Faster Opening](#)) introduced a connection-count field for this.

TCP is also vulnerable to sequence-number wraparound: arrival of an old duplicates from the *same* instance of the connection. However, if we take the MSL to be 60 seconds, sequence-number wrap requires sending  $2^{32}$  bytes in 60 seconds, which requires a data-transfer rate in excess of 500 Mbps. TCP offers a fix for this (Protection Against Wrapped Segments, or PAWS), but it was introduced relatively late; we return to this in [12.11 Anomalous TCP scenarios](#).

## 12.9 TIMEWAIT

The TIMEWAIT state is entered by whichever side initiates the connection close; in the event of a simultaneous close, both sides enter TIMEWAIT. It is to last for a time  $2 \times \text{MSL}$ , where MSL = Maximum Segment Lifetime is an agreed-upon value for the maximum lifetime on the Internet of an IP packet. Traditionally MSL was taken to be 60 seconds, but more modern implementations often assume 30 seconds (for a TIMEWAIT period of 60 seconds).

One function of TIMEWAIT is to solve the external-old-duplicates problem. TIMEWAIT requires that

between closing and reopening a connection, a long enough interval must pass that any packets from the first instance will disappear. After the expiration of the TIMEWAIT interval, an old duplicate cannot arrive.

A second function of TIMEWAIT is to address the lost-final-ACK problem ([11.3 Fundamental Transport Issues](#)). If host A sends its final ACK to host B and this is lost, then B will eventually retransmit *its* final packet, which will be its FIN. As long as A remains in state TIMEWAIT, it can appropriately reply to a retransmitted FIN from B with a duplicate final ACK. As with TFTP, it is possible (though unlikely) for the final ACK to be lost as well as all the retransmitted final FINs sent during the TIMEWAIT period; should this happen, one side thinks the connection closed normally while the other side thinks it did not. See exercise 12.0.

TIMEWAIT only blocks reconnections for which both sides reuse the same port they used before. If A connects to B and closes the connection, A is free to connect again to B using a different port at A's end.

Conceptually, a host may have many old connections to the same port simultaneously in TIMEWAIT; the host must thus maintain for each of its ports a list of all the remote  $\langle$ IP\_address, port $\rangle$  sockets currently in TIMEWAIT for that port. If a host is connecting as a client, this list likely will amount to a list of recently used ports; no port is likely to have been used twice within the TIMEWAIT interval. If a host is a server, however, accepting connections on a standardized port, and happens to be the side that initiates the active close and thus later goes into TIMEWAIT, then its TIMEWAIT list for that port can grow quite long.

Generally, busy servers prefer to be free from these bookkeeping requirements of TIMEWAIT, so many protocols are designed so that it is the client that initiates the active close. In the original HTTP protocol, version 1.0, the server sent back the data stream requested by the http GET message ([12.6.2 netcat again](#)), and indicated the end of this stream by closing the connection. In HTTP 1.1 this was fixed so that the client initiated the close; this required a new mechanism by which the server could indicate “I am done sending this file”. HTTP 1.1 also used this new mechanism to allow the server to send back multiple files over one connection.

In an environment in which many short-lived connections are made from host A to the same port on server B, port exhaustion – having all ports tied up in TIMEWAIT – is a theoretical possibility. If A makes 1000 connections per second, then after 60 seconds it has gone through 60,000 available ports, and there are essentially none left. While this rate is high, early Berkeley-Unix TCP implementations often made only about 4,000 ports available to clients; with a 120-second TIMEWAIT interval, port exhaustion would occur with only 33 connections per second.

If you use ssh to connect to a server and then issue the `netstat -a` command on your own host (or, more conveniently, `netstat -a | grep -i tcp`), you should see your connection in ESTABLISHED state. If you close your connection and check again, your connection should be in TIMEWAIT.

## 12.10 The Three-Way Handshake Revisited

As stated earlier in [12.3 TCP Connection Establishment](#), both sides choose an ISN; actual sequence numbers are the sum of the sender's ISN and the relative sequence number. There are two original reasons for this mechanism, and one later one ([12.10.1 ISNs and spoofing](#)). The original TCP specification, as clarified in [RFC 1122](#), called for the ISN to be determined by a special **clock**, incremented by 1 every 4 microseconds.

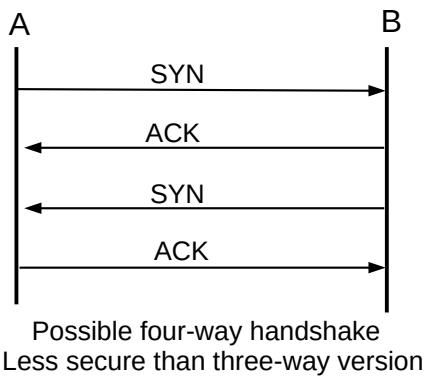
The most basic reason for using ISNs is to detect duplicate SYNs. Suppose A initiates a connection to B by sending a SYN packet. B replies with SYN+ACK, but this is lost. A then times out and retransmits its

SYN. B now receives A's second SYN while in state SYN\_RECEIVED. Does this represent an entirely new request (perhaps A has suddenly restarted), or is it a duplicate? If A uses the clock-driven ISN strategy, B can tell (*almost certainly*) whether A's second SYN is new or a duplicate: only in the latter case will the ISN values in the two SYNs match.

While there is no danger to data integrity if A sends a SYN, restarts, and sends the SYN again as part of a reopening the same connection, the arrival of a second SYN with a new ISN means that the original connection cannot proceed, because that ISN is now wrong. The receiver of the duplicate SYN should drop any connection state it has recorded so far, and restart processing the second SYN from scratch.

The clock-driven ISN also originally added a second layer of protection against external old duplicates. Suppose that A opens a connection to B, and chooses a clock-based ISN  $N_1$ . A then transfers M bytes of data, closed the connection, and reopens it with ISN  $N_2$ . If  $N_1 + M < N_2$ , then the old-duplicates problem *cannot occur*: all of the absolute sequence numbers used in the first instance of the connection are less than or equal to  $N_1 + M$ , and all of the absolute sequence numbers used in the second instance will be greater than  $N_2$ . In fact, early Berkeley-Unix implementations of the socket library often allowed a second connection meeting this ISN requirement to be reopened *before* TIMEWAIT would have expired; this potentially addressed the problem of port exhaustion. Of course, if the first instance of the connection transferred data faster than the ISN clock rate, that is at more than 250,000 bytes/sec, then  $N_1 + M$  would be greater than  $N_2$ , and TIMEWAIT would have to be enforced. But in the era in which TCP was first developed, sustained transfers exceeding 250,000 bytes/sec were not common.

The three-way handshake was extensively analyzed by Dalal and Sunshine in [DS78]. The authors noted that with a two-way handshake, the second side receives no confirmation that its ISN was correctly received. The authors also observed that a four-way handshake – in which the ACK of  $ISN_A$  is sent separately from  $ISN_B$ , as in the diagram below – could fail if one side restarted.



For this failure to occur, assume that after sending the SYN in line 1, with  $ISN_{A1}$ , A restarts. The ACK in line 2 is either ignored or not received. B now sends its SYN in line 3, but A interprets this as a new connection request; it will respond after line 4 by sending a fifth, SYN packet containing a different  $ISN_{A2}$ . For B the connection is now ESTABLISHED, and if B acknowledges this fifth packet but fails to update its record of A's ISN, the connection will fail as A and B would have different notions of  $ISN_A$ .

### 12.10.1 ISNs and spoofing

The clock-based ISN proved to have a significant weakness: it often allowed an attacker to guess the ISN a remote host might use. It did not help any that an early version of Berkeley Unix, instead of incrementing the

ISN 250,000 times a second, incremented it once a second, by 250,000 (plus something for each connection). By guessing the ISN a remote host would choose, an attacker might be able to mimic a local, trusted host, and thus gain privileged access.

Specifically, suppose host A trusts its neighbor B, and executes with privileged status commands sent by B; this situation was typical in the era of the `rhost` command. A authenticates these commands because the connection comes from B's IP address. The bad guy, M, wants to send packets to A so as to *pretend* to be B, and thus get a privileged command invoked. The connection only needs to be *started*; if the ruse is discovered after the command is executed, it is too late. M can easily send a SYN packet to A with B's IP address in the source-IP field; M can probably temporarily disable B too, so that A's SYN-ACK response, which is sent to B, goes unnoticed. What is harder is for M to figure out how to guess how to ACK ISN<sub>A</sub>. But if A generates ISNs with a slowly incrementing clock, M can guess the pattern of the clock with previous connection attempts, and can thus guess ISN<sub>A</sub> with a considerable degree of accuracy. So M sends SYN to A with B as source, A sends SYN-ACK to B containing ISN<sub>A</sub>, and M *guesses* this value and sends ACK(ISN<sub>A</sub>+1) to A, again with B listed in the IP header as source, followed by a single-packet command.

This TCP-layer IP-spoofing technique was first described by Robert T Morris in [\[RTM85\]](#); Morris went on to launch the [Internet Worm of 1988](#) using unrelated attacks. The IP-spoofing technique was used in the 1994 Christmas Day attack against UCSD, launched from Loyola's own apollo.it.luc.edu; the attack was associated with [Kevin Mitnick](#) though apparently not actually carried out by him. Mitnick was arrested a few months later.

[RFC 1948](#), in May 1996, introduced a technique for introducing a degree of randomization in ISN selection, while still ensuring that the same ISN would not be used twice in a row for the same connection. The ISN is to be the sum of the 4-μs clock, C(t), and a secure hash of the connection information as follows:

$$\text{ISN} = C(t) + \text{hash}(\text{local\_addr}, \text{local\_port}, \text{remote\_addr}, \text{remote\_port}, \text{key})$$

The `key` value is a random value chosen by the host on startup. While M, above, can poll A for its current ISN, and can probably guess the hash function and the first four parameters above, without knowing the key it cannot determine (or easily guess) the ISN value A would have sent to B. Legitimate connections between A and B, on the other hand, see the ISN increasing at the 4-μs rate.

[RFC 5925](#) addresses spoofing and related attacks by introducing an optional TCP authentication mechanism: the TCP header includes an option containing a secure hash ([22.6 Secure Hashes](#)) of the rest of the TCP header and a shared secret key. The need for key management limits when this mechanism can be used; the classic use case is BGP connections between routers ([10.6 Border Gateway Protocol, BGP](#)).

Another approach to the prevention of spoofing attacks is to ask sites and ISPs to refuse to forward outwards any IP packets with a source address not from within that site or ISP. If an attacker's ISP implements this, the attacker will be unable to launch spoofing attacks against the outside world. A concrete proposal can be found in [RFC 2827](#). Unfortunately, it has been (as of 2015) almost entirely ignored.

See also the discussion of SYN flooding at [12.3 TCP Connection Establishment](#), although that attack does not involve ISN manipulation.

## 12.11 Anomalous TCP scenarios

TCP, like any transport protocol, must address the transport issues in [11.3 Fundamental Transport Issues](#).

As we saw above, TCP addresses the Duplicate Connection Request (Duplicate SYN) issue by noting whether the ISN has changed. This is handled at the kernel level by TCP, versus TFTP's application-level (and rather desultory) approach to handing Duplicate RRQs.

TCP addresses Loss of Final ACK through TIMEWAIT: as long as the TIMEWAIT period has not expired, if the final ACK is lost and the other side resends its final FIN, TCP will still be able to reissue that final ACK. TIMEWAIT in this sense serves a similar function to TFTP's DALLY state.

External Old Duplicates, arriving as part of a previous instance of the connection, are prevented by TIMEWAIT, and may also be prevented by the use of a clock-driven ISN.

Internal Old Duplicates, from the *same* instance of the connection, that is, sequence number wraparound, is only an issue for bandwidths exceeding 500 Mbps: only at bandwidths above that can 4 GB be sent in one 60-second MSL. TCP implementations now address this with PAWS: Protection Against Wrapped Segments ([RFC 1323](#)). PAWS adds a 32-bit “timestamp option” to the TCP header. The granularity of the timestamp clock is left unspecified; one tick must be small enough that sequence numbers cannot wrap in that interval (eg less than 3 seconds for 10,000 Mbps), and large enough that the timestamps cannot wrap in time MSL. On Linux systems the timestamp clock granularity is typically 1 to 10 ms; measurements on the author's systems have been 4 ms. With timestamps, an old duplicate due to sequence-number wraparound can now easily be detected.

The PAWS mechanism also requires ACK packets to echo back the sender's timestamp, in addition to including their own. This allows senders to accurately measure round-trip times.

Reboots are a potential problem as the host presumably has no record of what aborted connections need to remain in TIMEWAIT. TCP addresses this on paper by requiring hosts to implement Quiet Time on Startup: no new connections are to be accepted for  $1^*MSL$ . No known implementations actually do this; instead, they assume that the restarting process itself will take at least one MSL. This is no longer as certain as it once was, but serious consequences have not ensued.

## 12.12 TCP Faster Opening

If a client wants to connect to a server, send a request and receive an immediate reply, TCP mandates one full RTT for the three-way handshake before data can be delivered. This makes TCP one RTT slower than UDP-based request-reply protocols. There have been periodic calls to allow TCP clients to include data with the first SYN packet and have it be delivered immediately upon arrival – this is known as **accelerated open**.

If there will be a series of requests and replies, the simplest fix is to **pipeline** all the requests and replies over one persistent connection; the one-RTT delay then applies only to the first request. If the pipeline connection is idle for a long-enough interval, it may be closed, and then reopened later if necessary.

An early accelerated-open proposal was **T/TCP**, or TCP for Transactions, specified in [RFC 1644](#). T/TCP introduced a **connection count** TCP option, called CC; each participant would include a 32-bit CC value in its SYN; each participant's own CC values were to be monotonically increasing. Accelerated open was allowed if the server side had the client's previous CC in a cache, and the new CC value was strictly greater than this cached value. This ensured that the new SYN was not a duplicate of an older SYN.

Unfortunately, this also bypasses the modest authentication of the client's IP address provided by the full three-way handshake, worsening the spoofing problem of [12.10.1 ISNs and spoofing](#). If malicious host M wants to pretend to be B when sending a privileged request to A, all M has to do is send a single SYN+Data

packet with an extremely large value for CC. Generally, the accelerated open succeeded as long as the CC value presented was larger than the value A had cached for B; it did not have to be larger by exactly 1.

The recent **TCP Fast Open** proposal, described in [RFC 7413](#), involves a secure “cookie” sent by the client as a TCP option; if a SYN+Data packet has a valid cookie, then the client has proven its identity and the data may be released immediately to the receiving application. Cookies are cryptographically secure, and are requested ahead of time from the server.

Because cookies have an expiration date and must be requested ahead of time, TCP Fast Open is not fundamentally faster from the connection-pipeline option, except that holding a TCP connection open uses more resources than simply storing a cookie. The likely application for TCP Fast Open is in accessing web servers. Web clients and servers already keep a persistent connection open for a while, but often “a while” here amounts only to several seconds; TCP Fast Open cookies could remain active for much longer.

One serious practical problem with TCP Fast Open is that some middleboxes ([7.7.2 Middleboxes](#)) remove TCP options they do not understand, or even block the connection attempt entirely.

### 12.13 Path MTU Discovery

TCP connections are more efficient if they can keep large packets flowing between the endpoints. Once upon a time, TCP endpoints included just 512 bytes of data in each packet that was not destined for local delivery, to avoid fragmentation. TCP endpoints now typically engage in **Path MTU Discovery** which almost always allows them to send larger packets; backbone ISPs are now usually able to carry 1500-byte packets. The **Path MTU** is the largest packet size that can be sent along a path without fragmentation.

The IPv4 strategy is to send an initial data packet with the IPv4 DONT\_FRAG bit set. If the ICMP message Frag\_Required/DONT\_FRAG\_Set comes back, or if the packet times out, the sender tries a smaller size. If the sender receives a TCP ACK for the packet, on the other hand, indicating that it made it through to the other end, it might try a larger size. Usually, the size range of 512-1500 bytes is covered by less than a dozen discrete values; the point is not to find the exact Path MTU but to determine a reasonable approximation rapidly.

IPv6 has no DONT\_FRAG bit. Path MTU Discovery over IPv6 involves the periodic sending of larger packets; if the ICMPv6 message Packet Too Big is received, a smaller packet size must be used. [RFC 1981](#) has details.

### 12.14 TCP Sliding Windows

TCP implements sliding windows, in order to improve throughput. Window sizes are measured in terms of bytes rather than packets; this leaves TCP free to packetize the data in whatever segment size it elects. In the initial three-way handshake, each side specifies the maximum window size it is willing to accept, in the **Window Size** field of the TCP header. This 16-bit field can only go to 64 kB, and a 1 Gbps  $\times$  100 ms bandwidth  $\times$  delay product is 12 MB; as a result, there is a TCP **Window Scale** option that can also be negotiated in the opening handshake. The scale option specifies a power of 2 that is to be multiplied by the actual Window Size value. In the Wireshark example above, the client specified a Window Size field of 5888 ( $= 4 \times 1472$ ) in the third packet, but with a Window Scale value of  $2^6 = 64$  in the first packet, for an

effective window size of  $64 \times 5888 = 256$  segments of 1472 bytes. The server side specified a window size of 5792 and a scaling factor of  $2^5 = 32$ .

TCP may either transmit a bulk stream of data, using sliding windows fully, or it may send slowly generated interactive data; in the latter case, TCP may never have even one full segment outstanding.

In the following chapter we will see that a sender frequently reduces the actual TCP window size, in order to avoid congestion; the window size included in the TCP header is known as the **Advertised Window Size**. On startup, TCP does not send a full window all at once; it uses a mechanism called “slow start”.

## 12.15 TCP Delayed ACKs

TCP receivers are allowed briefly to delay their ACK responses to new data. This offers perhaps the most benefit for interactive applications that exchange small packets, such as ssh and telnet. If A sends a data packet to B and expects an immediate response, delaying B’s ACK allows the receiving *application* on B time to wake up and generate that application-level response, which can then be sent together with B’s ACK. Without delayed ACKs, the kernel layer on B may send its ACK before the receiving application on B has even been scheduled to run. If response packets are small, that doubles the total traffic. The maximum ACK delay is 500 ms, according to [RFC 1122](#) and [RFC 2581](#).

For bulk traffic, delayed ACKs simply mean that the ACK traffic volume is reduced. Because ACKs are cumulative, one ACK from the receiver can in principle acknowledge multiple data packets from the sender. Unfortunately, acknowledging too many data packets with one ACK can interfere with the self-clocking aspect of sliding windows; the arrival of that ACK will then trigger a burst of additional data packets, which would otherwise have been transmitted at regular intervals. Because of this, the RFCs above specify that an ACK be sent, at a minimum, for every other data packet. For a discussion of how the sender should respond to delayed ACKs, see [13.2.1 TCP Reno Per-ACK Responses](#).

### Bandwidth Conservation

Delayed ACKs and the Nagle algorithm both originated in a bygone era, when bandwidth was in much shorter supply than it is today. In [RFC 896](#), John Nagle writes (in 1984, well before TCP Reno, [13 TCP Reno and Congestion Management](#)) “In general, we have not been able to afford the luxury of excess long-haul bandwidth that the ARPANET possesses, and our long-haul links are heavily loaded during peak periods. Transit times of several seconds are thus common in our network.” Today, it is unlikely that extra small packets would cause detectable, let alone significant, problems.

The TCP ACK-delay time can usually be adjusted globally as a system parameter. Linux offers a `TCP_QUICKACK` option, as a flag to `setsockopt()`, to disable delayed ACKs on a per-connection basis, but only until the next TCP system call. It must be invoked immediately after every receive operation to disable delayed ACKs entirely. This option is also not very portable.

The TSO option of [12.5 TCP Offloading](#), used at the receiver, can also reduce the number of ACKs sent. If every two arriving data packets are consolidated via TSO into a single packet, then the receiver will appear to the sender to be acknowledging every other data packet. The ACK delay introduced by TSO is, however, usually quite small.

## 12.16 Nagle Algorithm

Like delayed ACKs, the Nagle algorithm ([RFC 896](#)) also attempts to improve the behavior of interactive small-packet applications. It specifies that a TCP endpoint generating small data segments should queue them until either it accumulates a full segment's worth or receives an ACK for the previous batch of small segments. If the full-segment threshold is not reached, this means that only one (consolidated) segment will be sent per RTT.

As an example, suppose A wishes to send to B packets containing consecutive letters, starting with “a”. The application on A generates these every 100 ms, but the RTT is 501 ms. At T=0, A transmits “a”. The application on A continues to generate “b”, “c”, “d”, “e” and “f” at times 100 ms through 500 ms, but A does not send them immediately. At T=501 ms, ACK(“a”) arrives; at this point A transmits its backlog “bcdef”. The ACK for this arrives at T=1002, by which point A has queued “ghijk”. The end result is that A sends a fifth as many packets as it would without the Nagle algorithm. If these letters are generated by a user typing them with telnet, and the ACKs also include the echoed responses, then if the user pauses the echoed responses will very soon catch up.

The Nagle algorithm does not always interact well with delayed ACKs, or with user expectations; see exercises 10.0 and 10.5. It can usually be disabled on a per-connection basis, in the BSD socket library by calling `setsockopt()` with the `TCP_NODELAY` flag.

## 12.17 TCP Flow Control

It is possible for a TCP sender to send data faster than the receiver can process it. When this happens, a TCP receiver may reduce the advertised Window Size value of an open connection, thus informing the sender to switch to a smaller window size. This provides support for **flow control**.

The window-size reduction appears in the ACKs sent back by the receiver. A given ACK is not supposed to reduce the window size by so much that the upper end of the window gets smaller. A window might shrink from the byte range [20,000..28,000] to [22,000..28,000] but never to [20,000..26,000].

If a TCP receiver uses this technique to shrink the advertised window size to 0, this means that the sender may not send data. The receiver has thus informed the sender that, yes, the data was received, but that, no, more may not yet be sent. This corresponds to the `ACKWAIT` suggested in [6.1.3 Flow Control](#). Eventually, when the receiver is ready to receive data, it will send an ACK increasing the advertised window size again.

If the TCP sender has its window size reduced to 0, and the ACK from the receiver increasing the window is lost, then the connection would be deadlocked. TCP has a special feature specifically to avoid this: if the window size is reduced to zero, the sender sends dataless packets to the receiver, at regular intervals. Each of these “polling” packets elicits the receiver’s current ACK; the end result is that the sender will receive the eventual window-enlargement announcement reliably. These “polling” packets are regulated by the so-called **persist** timer.

## 12.18 Silly Window Syndrome

The silly-window syndrome is a term for a scenario in which TCP transfers only small amounts of data at a time. Because TCP/IP packets have a minimum fixed header size of 40 bytes, sending small packets uses

the network inefficiently. The silly-window syndrome can occur when either by the receiving application consuming data slowly or when the sending application generating data slowly.

As an example involving a slow-consuming receiver, suppose a TCP connection has a window size of 1000 bytes, but the receiving application consumes data only 10 bytes at a time, at intervals about equal to the RTT. The following can then happen:

- The sender sends bytes 1-1000. The receiving application consumes 10 bytes, numbered 1-10. The receiving TCP buffers the remaining 990 bytes and sends an ACK reducing the window size to 10, per [12.17 TCP Flow Control](#).
- Upon receipt of the ACK, the sender sends 10 bytes numbered 1001-1010, the most it is permitted. In the meantime, the receiving application has consumed bytes 11-20. The window size therefore remains at 10 in the next ACK.
- the sender sends bytes 1011-1020 while the application consumes bytes 21-30. The window size remains at 10.

The sender may end up sending 10 bytes at a time indefinitely. This is of no benefit to either side; the sender might as well send larger packets less often. The standard fix, set forth in [RFC 1122](#), is for the receiver to use its ACKs to keep the window at 0 until it has consumed one full packet's worth (or half the window, for small window sizes). At that point the sender is invited – by an appropriate window-size advertisement in the next ACK – to send another full packet of data.

The silly-window syndrome can also occur if the sender is *generating* data slowly, say 10 bytes at a time. The Nagle algorithm, above, can be used to prevent this, though for interactive applications sending small amounts of data in separate but closely spaced packets may actually be useful.

## 12.19 TCP Timeout and Retransmission

When TCP sends a packet containing user data (this excludes ACK-only packets), it sets a timeout. If that timeout expires before the packet data is acknowledged, it is retransmitted. Acknowledgments are sent for every arriving data packet (unless Delayed ACKs are implemented, [12.15 TCP Delayed ACKs](#)); this amounts to receiver-side retransmit-on-duplicate of [6.1.1 Packet Loss](#). Because ACKs are cumulative, and so a later ACK can replace an earlier one, lost ACKs are seldom a problem.

For TCP to work well for both intra-server-room and trans-global connections, with RTTs ranging from well under 1 ms to close to 1 second, the length of the timeout interval must *adapt*. TCP manages this by maintaining a running estimate of the RTT, EstRTT. In the original version, TCP then set TimeOut =  $2 \times \text{EstRTT}$  (in the literature, the TCP TimeOut value is often known as RTO, for Retransmission TimeOut). EstRTT itself was a running average of periodically measured SampleRTT values, according to

$$\text{EstRTT} = \alpha \times \text{EstRTT} + (1-\alpha) \times \text{SampleRTT}$$

for a fixed  $\alpha$ ,  $0 < \alpha < 1$ . Typical values of  $\alpha$  might be  $\alpha=1/2$  or  $\alpha=7/8$ . For  $\alpha$  close to 1 this is “conservative” in that EstRTT is slow to change. For  $\alpha$  closer to 0, EstRTT is more volatile.

There is a potential RTT measurement ambiguity: if a packet is sent twice, the ACK received could be in response to the first transmission or the second. The Karn/Partridge algorithm resolves this: on packet loss (and retransmission), the sender

- Doubles Timeout

- Stops recording SampleRTT
- Uses the doubled Timeout as EstRTT when things resume

Setting TimeOut =  $2 \times$  EstRTT proved too short during congestion periods and too long other times. Jacobson and Karels ([\[JK88\]](#)) introduced a way of calculating the TimeOut value based on the statistical variability of EstRTT. After each SampleRTT value was collected, the sender would also update EstDeviation according to

$$\text{SampleDev} = |\text{SampleRTT} - \text{EstRTT}|$$
$$\text{EstDeviation} = \beta \times \text{EstDeviation} + (1-\beta) \times \text{SampleDev}$$

for a fixed  $\beta$ ,  $0 < \beta < 1$ . Timeout was then set to  $\text{EstRTT} + 4 \times \text{EstDeviation}$ . EstDeviation is an estimate of the so-called *mean deviation*; 4 mean deviations corresponds (for normally distributed data) to about 5 *standard deviations*. If the SampleRTT values were normally distributed (which they are not), this would mean that the chance that a non-lost packet would arrive outside the TimeOut period is vanishingly small.

For further details, see [\[JK88\]](#) and [\[AP99\]](#).

Keeping track of when packets time out is usually handled by putting a record for each packet sent into a **timer list**. Each record contains the packet's timeout time, and the list is kept sorted by these times. Periodically, *eg* every 100 ms, the list is inspected and all packets with expired timeout are then retransmitted. When an ACK arrives, the corresponding packet timeout record is removed from the list. Note that this approach means that a packet's timeout processing may be slightly late.

## 12.20 KeepAlive

There is no reason that a TCP connection should not be idle for a long period of time; ssh/telnet connections, for example, might go unused for days. However, there is the turned-off-at-night problem: a workstation might telnet into a server, and then be shut off (not shut down gracefully) at the end of the day. The connection would now be half-open, but the server would not generate any traffic and so might never detect this; the connection itself would continue to tie up resources.

### KeepAlive in action

One evening long ago, when dialed up (yes, that long ago) into the Internet, my phone line disconnected while I was typing an email message in an ssh window. I dutifully reconnected, expecting to find my message in the file “dead.letter”, which is what would have happened had I been disconnected while using the even-older tty dialup. Alas, nothing was there. I reconstructed my email as best I could and logged off.

The next morning, there was my lost email in a file “dead.letter”, dated two hours after the initial crash! What had happened, apparently, was that the original ssh connection on the server side just hung there, half-open. Then, after two hours, KeepAlive kicked in, and aborted the connection. At that point ssh sent my mail program the HangUp signal, and the mail program wrote out what it had in “dead.letter”.

To avoid this, TCP supports an optional **KeepAlive** mechanism: each side “polls” the other with a dataless packet. The original [RFC 1122](#) KeepAlive timeout was 2 hours, but this could be reduced to 15 minutes. If a connection failed the KeepAlive test, it would be closed.

Supposedly, some TCP implementations are not exactly [RFC 1122](#)-compliant: either KeepAlives are enabled by default, or the KeepAlive interval is much smaller than called for in the specification.

## 12.21 TCP timers

To summarize, TCP maintains the following four kinds of timers. All of them can be maintained by a single timer list, above.

- **TimeOut:** a per-segment timer; TimeOut values vary widely
- $2 \times \text{MSL}$  **TIMEWAIT:** a per-connection timer
- **Persist:** the timer used to poll the receiving end when winsize = 0
- **KeepAlive,** above

## 12.22 Variants and Alternatives

One alternative to TCP is UDP with programmer-implemented timeout and retransmission; many RPC implementations ([11.5 Remote Procedure Call \(RPC\)](#)) do exactly this, with reasonable results. Within a LAN a static timeout of around half a second usually works quite well (unless the LAN has some tunneled links), and implementation of a simple timeout-retransmission mechanism is quite straightforward. Implementing adaptive timeouts as in [12.19 TCP Timeout and Retransmission](#) can, however, be a bit trickier. QUIC ([11.1 QUIC](#)) is an example of this strategy.

We here consider four other protocols. The first, MPTCP, is based on TCP itself. The second, SCTP, is a message-oriented alternative to TCP that is an entirely separate protocol. The last two, DCCP and QUIC, are attempts to create a TCP-like transport layer on top of UDP.

### 12.22.1 MPTCP

Multipath TCP, or MPTCP, allows connections to use multiple network interfaces on a host, either sequentially or simultaneously. MPTCP architectural principles are outlined in [RFC 6182](#); implementation details are in [RFC 6824](#).

To carry the actual traffic, MPTCP arranges for the creation of multiple standard-TCP **subflows** between the sending and receiving hosts; these subflows typically connect between different pairs of IP addresses on the respective hosts.

For example, a connection to a server can start using the client's wired Ethernet interface, and continue via Wi-Fi after the user has unplugged. If the client then moves out of Wi-Fi range, the connection might continue via a mobile network. Alternatively, MPTCP allows the parallel use of multiple Ethernet interfaces on both client and server for higher throughput.

MPTCP officially forbids the creation of multiple TCP connections between a single pair of interfaces in order to simulate Highspeed TCP ([15.5 Highspeed TCP](#)); [RFC 6356](#) spells out an MWTCP congestion-control algorithm to enforce this.

Suppose host A, with two interfaces with IP addresses  $A_1$  and  $A_2$ , wishes to connect to host B with IP addresses  $B_1$  and  $B_2$ . Connection establishment proceeds via the ordinary TCP three-way handshake, between one of A's IP addresses, say  $A_1$ , and one of B's,  $B_1$ . The SYN packets must each carry the MP\_CAPABLE TCP option, to signal one another that MPTCP is supported. As part of the MP\_CAPABLE option, A and B also exchange pseudorandom 64-bit connection keys, sent unencrypted; these will be used to sign later messages as in [22.6.1 Secure Hashes and Authentication](#). This first connection is the initial subflow.

Once the MPTCP initial subflow has been established, additional subflow connections can be made. Usually these will be initiated from the client side, here A, though the B side can also do this. At this point, however, A does not know of B's address  $B_2$ , so the only possible second subflow will be from  $A_2$  to  $B_1$ . New subflows will carry the MP\_JOIN option with their initial SYN packets, along with digital signatures signed by the original connection keys verifying that the new subflow is indeed part of this MPTCP connection.

At this point A and B can send data to one another using both connections simultaneously. To keep track of data, each side maintains a 64-bit data sequence number, DSN, for the data it sends; each side also maintains a mapping between the DSN and the subflow sequence numbers. For example, A might send 1000-byte blocks of data alternating between the  $A_1$  and  $A_2$  connections; the blocks might have DSN values 10000, 11000, 12000, 13000, .... The  $A_1$  subflow would then carry blocks 10000, 12000, etc, numbering these consecutively (perhaps 20000, 21000, ...) with its own sequence numbers. The sides exchange DSN mapping information with a DSS TCP option. This mechanism means that all data transmitted over the MWTCP connection can be delivered in the proper order, and that if one subflow fails, its data can be retransmitted on another subflow.

B can inform A of its second IP address,  $B_2$ , using the ADD\_ADDR option. Of course, it is possible that  $B_2$  is not directly reachable by A; for example, it might be behind a NAT router. But if  $B_2$  is reachable, A can now open two more subflows  $A_1$ — $B_2$  and  $A_2$ — $B_2$ .

All the above works equally well if either or both of A's addresses is behind a NAT router, simply because the NAT router is able to properly forward the subflow TCP connections. Addresses sent from one host to another, such as B's transmission of its address  $B_2$ , may be rendered invalid by NAT, but in this case A's attempt to open a connection to  $B_2$  simply fails.

Generally, hosts can be configured to use multiple subflows in parallel, or to use one interface only as a backup, when the primary interface is unplugged or out of range. APIs have been proposed that allow an control over MPTCP behavior on a per-connection basis.

### 12.22.2 SCTP

The Stream Control Transmission Protocol, SCTP, is an entirely separate protocol from TCP, running directly above IP. It is, in effect, a message-oriented alternative to TCP: an application writes a sequence of messages and SCTP delivers each one as a unit, fragmenting and reassembling it as necessary. Like TCP, SCTP is connection-oriented and reliable. SCTP uses a form of sliding windows, and, like TCP, adjusts the window size to manage congestion.

An SCTP connection can support multiple **message streams**; the exact number is negotiated at startup. A retransmission delay in one stream never blocks delivery in other streams. Within each stream, SCTP messages are sequentially numbered, and are normally delivered in order of message number. A receiver can request, however, to receive messages immediately upon successful delivery, that is, potentially out of order. Either way, the data within each message is guaranteed to be delivered in order and without loss.

Internally, message data is divided into SCTP **chunks** for inclusion in packets. One SCTP packet can contain data chunks from different messages and different streams; packets can also contain control chunks.

Messages themselves can be quite large; there is no set limit. Very large messages may need to be received in multiple system calls (*e.g.* calls to `recvmsg()`).

SCTP supports an MPTCP-like feature by which each endpoint can use multiple network interfaces.

SCTP connections are set up using a four-way handshake, versus TCP's three-way handshake. The extra packet provides some protection against so-called SYN flooding ([12.3 TCP Connection Establishment](#)). The central idea is that if client A initiates a connection request with server B, then B allocates no resources to the connection until after B has received a response to its own message to A. This means that, at a minimum, A is a real host with a real IP address.

The full four-way handshake between client A and server B is, in brief, as follows:

- A sends B an INIT chunk (corresponding to SYN), along with a pseudorandom Tag<sub>A</sub>.
- B sends A an INIT ACK, with Tag<sub>B</sub> and a **state cookie**. The state cookie contains all the information B needs to allocate resources to the connection, and is digitally signed ([22.6.1 Secure Hashes and Authentication](#)) with a key known only to B. Crucially, B does **not** at this point allocate any resources to the incipient connection.
- A returns the state cookie to B in a COOKIE ECHO packet.
- B enters the ESTABLISHED state and sends a COOKIE ACK to A. Upon receipt, A enters the ESTABLISHED state.

When B receives the COOKIE ECHO, it verifies the signature. At this point B knows that it sent the cookie to A and received a response, so A must exist. Only then does B allocate memory resources to the connection. Spoofed INITs in the first step cost B essentially nothing.

The Tag<sub>A</sub> and Tag<sub>B</sub> in the first two packets are called **verification tags**. From this point on, B will include Tag<sub>A</sub> in every packet it sends to A, and vice-versa. Although these tags are sent unencrypted, they nonetheless make it much harder for an attacker to inject data into the connection.

Data can be included in the third and fourth packets above; *i.e.* A can begin sending data after one RTT.

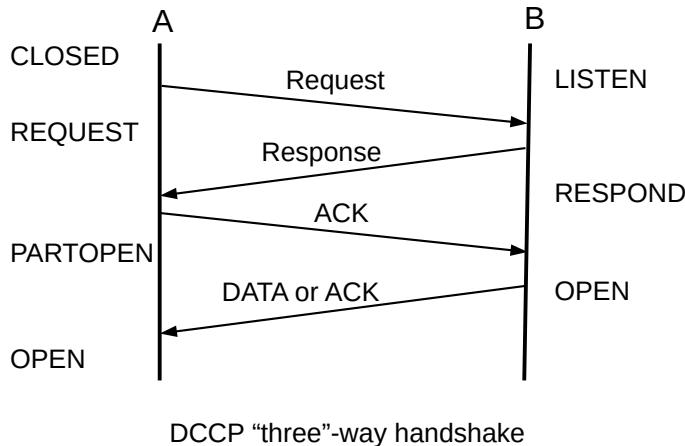
Unfortunately for potential SCTP applications, few if any NAT routers recognize SCTP; this limits the use of SCTP to Internet paths along which NAT is not used. In principle SCTP could simplify delivery of web pages, transmitting one page component per message, but lack of NAT support makes this infeasible. SCTP is also blocked by some middleboxes ([7.7.2 Middleboxes](#)) on the grounds that it is an unknown protocol, and therefore suspect. While this is not quite as common as the NAT problem, it is common enough to prevent by itself the widespread adoption of SCTP in the general Internet. SCTP is widely used for telecommunications signaling, both within and between providers, where NAT and recalcitrant middleboxes can be banished.

### 12.22.3 DCCP

As we saw in [11.1.2 DCCP](#), DCCP is a UDP-based transport protocol that supports, among other things, connection establishment. While it is used much less often than TCP, it provides an alternative example of how transport can be done.

DCCP defines a set of distinct packet types, rather than TCP's independent packet flags; this disallows unforeseen combinations such as TCP SYN+RST. Connection establishment involves Request and Respond; data transmission involves Data, ACK and DataACK, and teardown involves CloseReq, Close and Reset. While one cannot have, for example, a Respond+ACK, Respond packets do carry an acknowledgment field.

Like TCP, DCCP uses a three-way handshake to open a connection; here is a diagram:

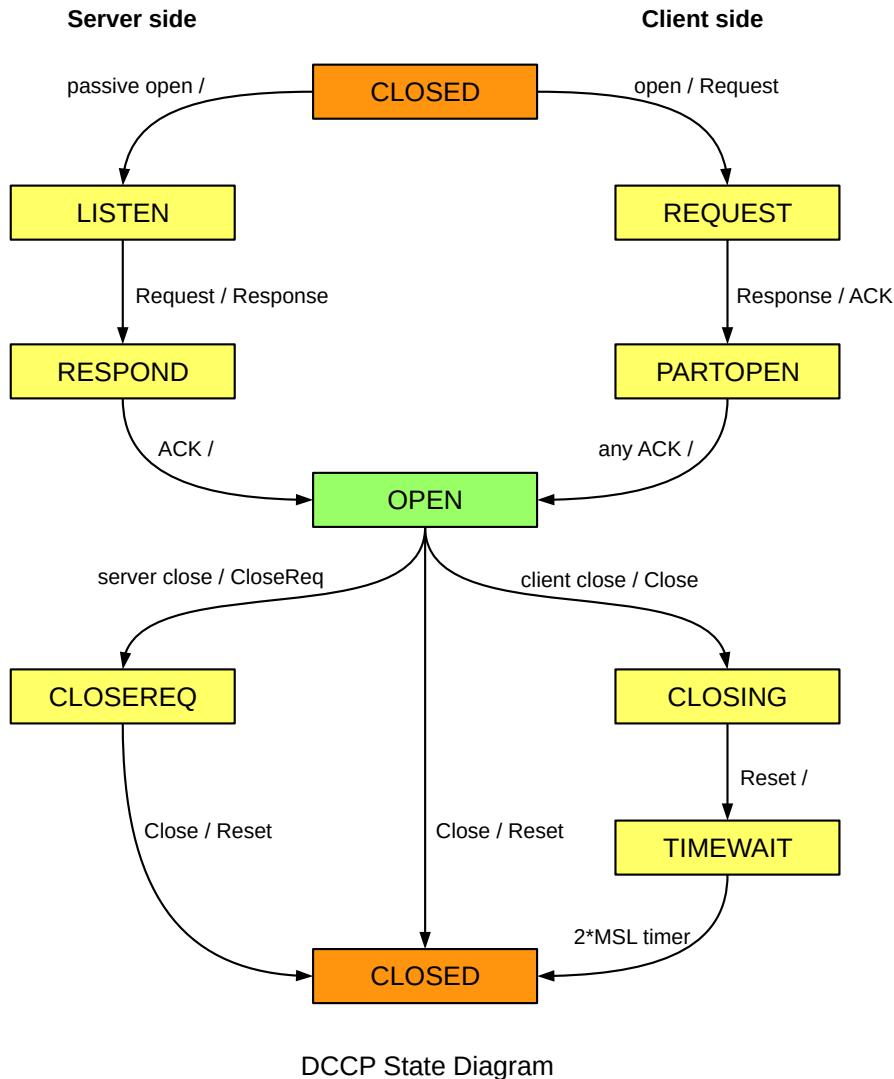


The fourth packet, DATA or ACK, is not considered part of the handshake itself.

The OPEN state corresponds to TCP's ESTABLISHED state. Like TCP, each side chooses an ISN (not shown in the diagram). Because packet delivery is not reliable, and because ACKs are not cumulative, the client remains in PARTOPEN state until it has confirmed that the server has received its ACK of the server's Response. While in state PARTOPEN, the client can send ACK and DataACK but not ACK-less Data packets.

Packets are numbered sequentially. The numbering includes all packets, not just Data packets, and is by packet rather than by byte.

The DCCP state diagram is shown below. It is simpler than the TCP state diagram because DCCP does not support simultaneous opens.



To close a connection, one side sends Close and the other responds with Reset. Reset is used for normal close as well as for exceptional conditions. Because whoever sends the Close is then stuck with TIMEWAIT, the server side may send CloseReq to ask the client to send Close.

There are also two special packet formats, Sync and SyncAck, for resynchronizing sequence numbers after a burst of lost packets.

The other major TCP-like feature supported by DCCP is congestion control; see [14.6.3 DCCP Congestion Control](#).

#### 12.22.4 QUIC Revised

Like DCCP, QUIC is also a UDP-based transport protocol, aimed rather squarely at HTTP plus TLS ([22.10.2 TLS](#)). The fundamental goal of QUIC is to provide TLS encryption protection with as little overhead as possible, in a manner that competes fairly with TCP in the presence of congestion. Opening a QUIC connection, encryption included, takes a single RTT. QUIC can also be seen, however, as a complete rewrite

of TCP from the ground up; a reading of specific features sheds quite a bit of light on how the corresponding TCP features have fared over the past thirty-odd years. QUIC is currently (2018) documented in a set of Internet Drafts:

- Transport basics: [draft-ietf-quic-transport](#)
- Loss and congestion management: [draft-ietf-quic-recovery](#)
- TLS encryption over QUIC: [draft-ietf-quic-tls](#)
- HTTP over QUIC: [draft-ietf-quic-http](#)

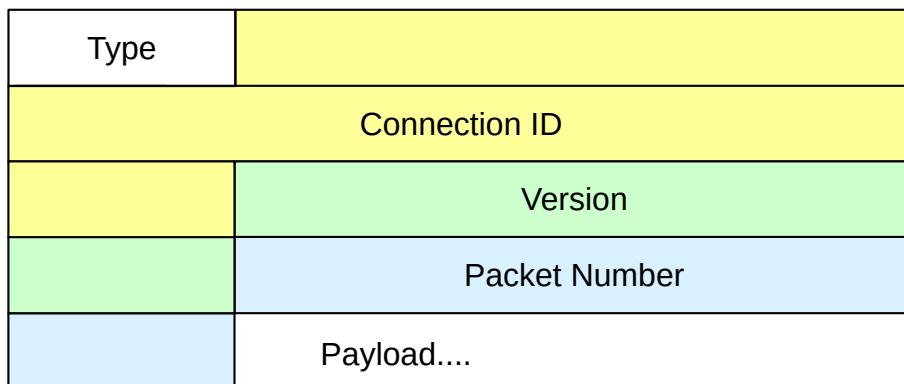
The last Internet Draft above was retitled in November 2018 as [Hypertext Transfer Protocol Version 3 \(HTTP/3\)](#); that is, it proposes that QUIC should become HTTP version 3. If and when this happens, it may represent the beginning of the end for TCP, given that most Internet connections are for HTTP or HTTPS. Still, many TCP design issues ([13 TCP Reno and Congestion Management](#), [14 Dynamics of TCP](#), [15 Newer TCP Implementations](#)) carry over very naturally to QUIC; a shift from TCP to QUIC should best be viewed as evolutionary.

The design of QUIC was influenced by the fate of SCTP above; the latter, as a new protocol above IP, was often blocked by overly security-conscious middleboxes ([7.7.2 Middleboxes](#)).

The fact that the QUIC layer resides within an application (or within a library) rather than within the kernel has meant that QUIC is able to evolve much faster than TCP. The long-term consequences of having the transport layer live outside the kernel are not yet completely clear, however; it may, for example, make it easier for users to install unfair congestion-management schemes.

### 12.22.4.1 Headers

We will start with the QUIC header. While there are some alternative forms, the basic header is diagrammed below, with a 1-byte Type field, an 8-byte Connection ID, and 4-byte Version and Packet Number fields.



Typical QUIC Long Header

Perhaps the most striking thing about this header is that 4-byte alignment – used consistently in the IPv4, IPv6, UDP and TCP headers – has been completely abandoned. On most contemporary processors, the performance advantages of alignment are negligible; see the last paragraph at [7.1 The IPv4 Header](#).

IP packets are identified as such by the Ethernet type field, and TCP and UDP packets are identified as such by the IPv4-header Protocol field. But QUIC packets are *not* identified as such by any flag in the preceding IP or UDP headers; there is in fact no place in those headers for a QUIC marker to go. QUIC appears to an observer as just another form of UDP traffic. This acts as a form of middlebox defense; QUIC packets cannot be identified as such in isolation. Wireshark, sidebar below, identifies QUIC packets by looking at the whole history of the connection, and even then must make some (educated) guesses. Middleboxes could do that too, but it would take work.

The initial Connection ID consists of 64 random bits chosen by the client. The server, upon accepting the connection, may change the Connection ID; at that point the Connection ID is fixed for the lifetime of the connection. The Connection ID may be omitted for packets whose connection can be determined from the associated IP address and port values; this is signaled by the Type field. The Connection ID can also be used to migrate a connection to a different IP address and port, as might happen if a mobile device moves out of range of Wi-Fi and the mobile-data plan continues the communication. This may also happen if a connection passes through a NAT router. The NAT forwarding entry may time out (see the comment on UDP and inactivity at [7.7 Network Address Translation](#)), and the connection may be assigned a different outbound UDP port if it later resumes. QUIC uses the Connection ID to recognize that the reassigned connection is still the same one as before.

The Version field gets dropped as soon as the version is negotiated. As part of the version negotiation, a packet might have multiple version fields. Such packets put a *random* value into the low-order seven bits of the Type field, as a prevention against middleboxes' blocking unknown types. This way, aggressive middlebox behavior should be discovered early, before it becomes widespread.

### QUIC-watching

QUIC packets can be observed in [Wireshark](#) by using the filter string “quic”. To generate QUIC traffic, use a [Chromium-based browser](#) and go to a Google-operated site, say, [google.com](http://google.com). Often the only non-encrypted fields are the Type field and the packet number. It may be necessary to enable QUIC in the browser, done in Chrome via chrome://flags; see also chrome://net-internals.

The packet number can be reduced to one or two bytes once the connection is established; this is signaled by the Type field. Internally, QUIC uses packet numbers in the range 0 to  $2^{62}$ ; these internal numbers are not allowed to wrap around. The low-order 32 bits (or 16 bits or 8 bits) of the internal number are what is transmitted in the packet header. A packet receiver infers the high-order bits from the most recent acknowledgment.

The initial packet number is to be chosen randomly in the range 0 to  $2^{32}-1025$ . This corresponds to TCP's use of Initial Sequence Numbers.

Use of 16-bit or 8-bit transmitted packet numbers is restricted to cases where there can be no ambiguity. At a minimum, this means that the number of outstanding packets (the QUIC winsize) cannot exceed  $2^7-1$  for 8-bit packet numbering or  $2^{15}-1$  for 16-bit packet numbering. These maximum winsizes represent the ideal case where there is no packet reordering; smaller values are likely to be used in practice. (See [6.5 Exercises](#), exercise 9.0.)

#### 12.22.4.2 Frames and streams

Data in a QUIC packet is partitioned into one or more **frames**. Each frame's data is prefixed by a simple frame header indicating its length and type. Some frames contain management information; frames containing higher-layer data are called STREAM frames. Each frame must be fully contained in one packet.

The application's data can be divided into multiple **streams**, depending on the application requirements. This is particularly useful with HTTP, as a client may request a large number of different resources (html, images, javascript, *etc*) simultaneously. Stream data is contained in STREAM frames. Streams are numbered, with Stream 0 reserved for the TLS cryptographic handshake. The HTTP/2 protocol has introduced its own notion of streams; these map neatly onto QUIC streams.

The two low-order bits of each stream number indicate whether the stream was initiated by the client or by the server, and whether it is bi- or uni-directional. This design decision means that either side can create a stream and send data on it immediately, *without negotiation*; this is important for reducing unnecessary RTTs.

Each individual stream is guaranteed in-order delivery, but there are no ordering guarantees between different streams. Within a packet, the data for a particular stream is contained in a frame for that stream.

One packet can contain stream frames for multiple streams. However, if a packet is lost, streams that have frames contained in that packet are blocked until retransmission. Other streams can continue without interruption. This creates an incentive for keeping separate streams in separate packets.

Stream frames contain the byte offset of the frame's block of stream data (starting from 0), to enable in-order stream reassembly. TCP, as we have seen, uses this byte-numbering approach exclusively, though starting with the Initial Sequence Number rather than zero. QUIC's stream-level numbering by byte is unrelated to its top-level numbering by packet.

In addition to stream frames, there are a large number of management frames. Here are a few of them:

- RST\_STREAM: like TCP RST, but for one stream only.
- MAX\_DATA: this corresponds to the TCP advertised window size. As with TCP, it can be reduced to zero to pause the flow of data and thereby implement flow control. There is also a similar MAX\_STREAM\_DATA, applying per stream.
- PING and PONG: to verify that the other endpoint is still responding. These serve as the equivalent of TCP KEEPALIVEs, among other things.
- CONNECTION\_CLOSE and APPLICATION\_CLOSE: these initiate termination of the connection; they differ only in that a CONNECTION\_CLOSE might be accompanied by a QUIC-layer error or explanation message while an APPLICATION\_CLOSE might be accompanied by, say, an HTTP error/explanation message.
- PAD: to pad out the packet to a larger size.
- ACK: for acknowledgments, below.

#### 12.22.4.3 Acknowledgments

QUIC assigns a new, sequential packet number (the `Packet ID`) to every packet, including retransmissions. TCP, by comparison, assigns sequence numbers to each byte. (QUIC stream frames do number data

by byte, as noted above.)

Lost QUIC packets are retransmitted, but with a new packet number. This makes it impossible for a receiver to send cumulative acknowledgments, as lost packets will never be acknowledged. The receiver handles this as below. At the sender side, the sender maintains a list of packets it has sent that are both unacknowledged and also not known to be lost. These represent the packets **in flight**. When a packet is retransmitted, its old packet number is removed from this list, as lost, and the new packet number replaces it.

To the extent possible given this retransmission-renumbering policy, QUIC follows the spirit of sliding windows. It maintains a state variable `bytes_in_flight`, corresponding to TCP's `winsize`, listing the total size of all the packets in flight. As with TCP, new acknowledgments allow new transmissions.

Acknowledgments themselves are sent in special acknowledgment frames. These begin with the number of the highest packet received. This is followed by a list of pairs, as long as will fit into the frame, consisting of the length of the next block of contiguous packets received followed by the length of the intervening gap of packets *not* received. The TCP Selective ACK ([13.6 Selective Acknowledgments \(SACK\)](#)) option is similar, but is limited to three blocks of received packets. It is quite possible that some of the gaps in a QUIC ACK frame refer to lost packets that were long since retransmitted with new packet numbers, but this does not matter.

The sender is allowed to skip packet numbers occasionally, to prevent the receiver from trying to increase throughput by acknowledging packets not yet received. Unlike with TCP, acknowledging an unsent packet is considered to be a fatal error, and the connection is terminated.

As with TCP, there is a delayed-ACK timer, but, while TCP's is typically 250 ms, QUIC's is 25 ms. QUIC also includes in each ACK frame the receiver's best estimate of the elapsed time between arrival of the most recent packet and the sending of the ACK it triggered; this allows the sender to better estimate the RTT. The primary advantage of the design decision not to reuse packet IDs is that there is never any ambiguity as to a retransmitted packet's RTT, as there is in TCP ([12.19 TCP Timeout and Retransmission](#)). Note, however, that because QUIC runs in a user process and not the kernel, it may not be able to respond immediately to an arriving packet, and so the time-delay estimate may be slightly short.

ACK frames are not themselves acknowledged. This means that, in a one-way data flow, the receiver may have no idea if its ACKs are getting through (a TCP receiver may be in the same situation). The QUIC receiver may send a PING frame to the sender, which will respond not only with a matching PONG frame but also an ACK frame acknowledging the receiver's recent acknowledgment packets.

QUIC adjusts its `bytes_in_flight` value to manage congestion, much as TCP manages its `winsize` (or more properly its `cwnd`, [13 TCP Reno and Congestion Management](#)) for the same purpose. Specifically, QUIC attempts to mimic the congestion response of TCP Cubic, [15.15 TCP CUBIC](#), and so should in theory compete fairly with TCP Cubic connections. However, it is straightforward to arrange for QUIC to model the behavior of any other flavor of TCP ([15 Newer TCP Implementations](#)).

#### 12.22.4.4 Connection handshake and TLS encryption

The opening of a QUIC connection makes use of the TLS handshake, [22.10.2 TLS](#), specifically TLS v1.3, [22.10.2.4.3 TLS version 1.3](#). A client wishing to connect sends a QUIC Initial packet, containing the TLS ClientHello message. The server responds (with a ServerHello) in a QUIC Handshake packet. (There is also a Retry packet, for special situations.) The TLS negotiation is contained in QUIC's

Stream 0. While the TLS and QUIC handshake rules are rather precise, there is as yet no formal state-diagram description of connection opening.

The `Initial` packet also contains a set of **QUIC transport parameters** declared unilaterally by the client; the server makes a similar declaration in its response. These parameters include, among other things, the maximum packet size, the connection's idle timeout, and initial value for `MAX_DATA`, above.

An important feature of TLS v1.3 is that, if the client has connected to the server previously and still has the key negotiated in that earlier session, it can use that old key to send an encrypted application-layer request (in a STREAM frame) immediately following the `Initial` packet. This is called **0-RTT** protection (or encryption). The advantage of this is that the client may receive an answer from the server within a single RTT, versus four RTTs for traditional TCP (one for the TCP three-way handshake, two for TLS negotiation, and one for the application request/reply). As discussed at [22.10.2.4.4 TLS v1.3 0-RTT mode](#), requests submitted with 0-RTT protection must be idempotent, to prevent replay attacks.

Once the server's first `Handshake` packet makes it back to the client, the client is in possession of the key negotiated by the new session, and will encrypt everything using that going forward. This is known as the **1-RTT** key, and all further data is said to be 1-RTT protected. The negotiated key is initially calculated by the TLS layer, which then exports it to QUIC. The QUIC layer then encrypts the entire data portion of its packets, using the format of [RFC 5116](#).

The QUIC header is not encrypted, but is still covered by an authentication checksum, making it impossible for middleboxes to rewrite anything. Such rewriting has been observed for TCP, and has sometimes complicated TCP evolution.

The type field of a QUIC packet contains a special code to mark 0-RTT data, ensuring that the receiver will know what level of protection is in effect.

When a QUIC server receives the `ClientHello` and sends off its `ServerHello`, it has not yet received any evidence that the client "owns" the IP address it claims to have; that is, that the client is not spoofing its IP address ([12.10.1 ISNs and spoofing](#)). Because of the idempotency restriction on responses to 0-RTT data, the server cannot give away privileges if spoofed in this way by a client. The server may, however, be an unwitting participant in a **traffic-amplification** attack, if the real client can trigger the sending by the server to a spoofed client of a larger response than the real client sends directly. The solution here is to require that the QUIC `Initial` packet, containing the `ClientHello`, be at least 1200 bytes. The server's `Handshake` response is likely to be smaller, and so represents no amplification of traffic.

To close the connection, one side sends a `CONNECTION_CLOSE` or `APPLICATION_CLOSE`. It may continue to send these in response to packets from the other side. When the other side receives the `CLOSE` packet, it should send its own, and then enter the so-called **draining** state. When the initiator of the close receives the other side's echoed `CLOSE`, it too will enter the draining state. Once in this state, an endpoint may not send any packets. The draining state corresponds to TCP's `TIMEWAIT` ([12.9 TIMEWAIT](#)), for the purpose of any lost final ACKs; it should last three RTT's. There is no need of a `TIMEWAIT` analog to prevent old duplicates, as a second QUIC connection will select a new Connection ID.

QUIC connection closing has no analog of TCP's feature in which one side sends FIN and the other continues to send data indefinitely, [12.7.1 Closing a connection](#). This use of FIN, however, is allowed in bidirectional streams; the per-stream (and per-direction) FIN bit lives in the stream header. Alternatively, one side can send its request and close its stream, and the other side can then answer on a different stream.

## 12.23 Epilog

At this point we have covered the basic mechanics of TCP, but have one important topic remaining: how TCP manages its window size so as to limit congestion, while maintaining fairness. This turns out to be complex, and will be the focus of the next three chapters.

## 12.24 Exercises

*Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercise 4.5 is distinct, for example, from exercises 4.0 and 5.0.*

1.0. Experiment with the TCP version of simplex-talk. How does the server respond differently with threading enabled and without, if two simultaneous attempts to connect are made, from two different client instances?

2.0. Trace the states visited if nodes A and B attempt to create a TCP connection by *simultaneously* sending each other SYN packets, that then cross in the network. Draw the ladder diagram, and label the states on each side. Hint: there should be two pairs of crossing packets. A SYN+ACK counts as an ACK.

2.5. Suppose nodes A and B are each behind their own NAT firewall ([7.7 Network Address Translation](#)).

A —— NAT\_A —— Internet —— NAT\_B —— B

A and B attempt to connect to one another simultaneously, using TCP. A sends to the public IPv4 address of NAT\_B, and vice-versa for B. Assume that neither NAT\_A nor NAT\_B changes the port numbers in outgoing packets, at least for the packets involved in this connection attempt. Show that the connection succeeds.

3.0. When two nodes A and B simultaneously attempt to connect to one another using the OSI TP4 protocol, two bidirectional network connections are created (rather than one, as with TCP). If TCP had instead chosen the TP4 semantics here, what would have to be added to the TCP header? Hint: if a packet from  $\langle A, \text{port1} \rangle$  arrives at  $\langle B, \text{port2} \rangle$ , how would we tell to which of the two possible connections it belongs?

4.0. Simultaneous connection initiations are rare, but simultaneous connection termination is relatively common. How do two TCP nodes negotiate the simultaneous sending of FIN packets to one another? Draw the ladder diagram, and label the states on each side. Which node goes into TIMEWAIT state? Hint: there should be two pairs of crossing packets.

4.5. The state diagram at [12.7 TCP state diagram](#) shows a dashed path from FIN\_WAIT\_1 to TIMEWAIT on receipt of FIN+ACK. All FIN packets contain a valid ACK field, but that is not what is meant here. Under what circumstances is this direct arc from FIN\_WAIT\_1 to TIMEWAIT taken? Explain why this arc can never be used during simultaneous close. Hint: consider the ladder diagram of a “normal” close.

5.0. (a) Suppose you see multiple connections on your workstation in state FIN\_WAIT\_1. What is likely going on? Whose fault is it? | (b). What might be going on if you see connections languishing in state FIN\_WAIT\_2?

6.0. Suppose that, after downloading a file, the client host is unplugged from the network, so it can send no further packets. The server’s connection is still in the ESTABLISHED state. In each case below, use the TCP state diagram to list all states that are reachable by the server.

- (a). Before being unplugged, the client was in state ESTABLISHED; *ie* it had *not* sent the first FIN.
- (b). Before being unplugged the client had sent its FIN, and moved to FIN\_WAIT\_1.

Eventually, the server connection here would in fact transition to CLOSED due to repeated timeouts. For this exercise, though, assume only transitions explicitly shown in the state diagram are allowed.

6.5. In [12.3 TCP Connection Establishment](#) we noted that RST packets had to have a valid SYN value, but that “[RFC 793](#) does not require the RST packet’s ACK value to match”. There is an exception for RST packets arriving at state SYN-SENT: “the RST is acceptable if the ACK field acknowledges the SYN”. Explain the reasoning behind this exception.

7.0. Suppose A and B create a TCP connection with  $ISN_A=20,000$  and  $ISN_B=5,000$ . A sends three 1000-byte packets (Data1, Data2 and Data3 below), and B ACKs each. Then B sends a 1000-byte packet DataB to A and terminates the connection with a FIN. In the table below, fill in the SEQ and ACK fields for each packet shown.

A sends	B sends
SYN, $ISN_A=20,000$	
	SYN, $ISN_B=5,000$ , ACK=_____
ACK, SEQ=_____, ACK=_____	
Data1, SEQ=_____, ACK=_____	
	ACK, SEQ=_____, ACK=_____
Data2, SEQ=_____, ACK=_____	
	ACK, SEQ=_____, ACK=_____
Data3, SEQ=_____, ACK=_____	
	ACK, SEQ=_____, ACK=_____
	DataB, SEQ=_____, ACK=_____
ACK, SEQ=_____, ACK=_____	
	FIN, SEQ=_____, ACK=_____

8.0. Suppose you are downloading a large file, and there is a progress bar showing how much of the file has been downloaded. For definiteness, assume the progress bar moves 1 mm per MB, the throughput averages 0.5 MB per second (so the progress bar advances at a rate of 0.5 mm/sec), and the winsize is 5 MB.

A packet is now lost, and is retransmitted after a timeout. What will happen to the progress bar? If someone measured the progress bar at two times 1 second apart, just before and just after the lost packet arrived, what value would they calculate for the throughput?

9.0. Suppose you are creating software for a streaming-video site. You want to limit the video read-ahead – the gap between how much has been downloaded and how much the viewer has actually watched – to approximately 1 MB; the server should pause in sending when necessary to enforce this. On the other hand, you do want the receiver to be able to read ahead by up to this much. You should assume that the TCP connection throughput will be higher than the actual video-data-consumption rate.

- (a). Suppose the TCP window size happens to be exactly 1 MB. If the receiver simply reads each video frame from the TCP connection, displays it, and then pauses briefly before reading the next frame in

accordance with the frame rate, explain how the flow-control mechanism of [12.17 TCP Flow Control](#) will achieve the desired effect.

(b). Applications, however, cannot control their TCP window size. What support would you have to add to the video-transfer *application* to allow it to read ahead by 1 MB but not to exceed this? Hint: both client and server sides of the application will have to implement something to enable this feature.

10.0. A user moves the computer mouse and sees the mouse-cursor's position updated on the screen. Suppose the mouse-position updates are being transmitted over a TCP connection with a relatively long RTT. The user attempts to move the cursor to a specific point. How will the user perceive the mouse's motion

- (a). with the Nagle algorithm
- (b). without the Nagle algorithm

10.5. Host A sends two single-byte packets, one containing “x” and the other containing “y”, to host B. A implements the Nagle algorithm and B implements delayed ACKs, with a 500 ms maximum delay. The RTT is negligible. How long does the transmission take? Draw a ladder diagram.

11.0. Suppose you have fallen in with a group that wants to add to TCP a feature so that, if A and B1 are connected, then B1 can **hand off** its connection to a different host B2; the end result is that A and B2 are connected and A has received an uninterrupted stream of data. Either A or B1 can initiate the handoff.

- (a). Suppose B1 is the host to send the final FIN (or HANOFF) packet to A. How would you handle appropriate analogues of the TIMEWAIT state for host B1? Does the fact that A is continuing the connection, just not with B1, matter?
- (b). Now suppose A is the party to send the final FIN/HANOFF, to B1. What changes to TIMEWAIT would have to be made at A's end? Note that A may potentially hand off the connection again and again, *eg* to B3, B4 and then B5.

12.0. Suppose A connects to B via TCP, and sends the message “Attack at noon”, followed by FIN. Upon receiving this, B is sure it has received the entire message.

- (a). What can A be sure of upon receiving B's own FIN+ACK?
- (b). What can B be sure of upon receiving A's final ACK?
- (c). What is A not absolutely sure of after sending its final ACK?

13.0. Host A connects to the Internet via Wi-Fi, receiving IPv4 address 10.0.0.2, and then opens a TCP connection *conn1* to remote host B. After *conn1* is established, A's Ethernet cable is plugged in. A's Ethernet interface receives IP address 10.0.0.3, and A automatically selects this new Ethernet connection as its default

route. **Assume** that A now starts using 10.0.0.3 as the source address of packets it sends as part of *conn1* (contrary to [RFC 1122](#)).

**Assume also** that A's TCP implementation is such that when a packet arrives from  $\langle B_{IP}, B_{port} \rangle$  to  $\langle A_{IP}, A_{port} \rangle$  and this socketpair is to be matched to an existing TCP connection, the field  $A_{IP}$  is allowed to be any of A's IP addresses (that is, either 10.0.0.2 or 10.0.0.3); it does not have to match the IP address with which the connection was originally negotiated.

- (a). Explain why *conn1* will now fail, as soon as any packet is sent from A. Hint: the packet will be sent from 10.0.0.3. What will B send in response? In light of the second assumption, how will A react to B's response packet?

(The author regularly sees connections appear to fail this way. Perhaps some justification for this behavior is that, at the time of establishment of *conn1*, A was not yet multihomed.)

- (b). Now suppose all four fields of the socketpair ( $\langle B_{IP}, B_{port} \rangle, \langle A_{IP}, A_{port} \rangle$ ) are used to match an incoming packet to its corresponding TCP connection. The connection *conn1* still fails, though not as immediately. Explain what happens.

See also [7.9.5 ARP and multihomed hosts](#), [7 IP version 4](#) exercise 11.0, and [9 Routing-Update Algorithms](#) exercise 13.0.

14.0. Modify the simplex-talk server of [12.6 TCP simplex-talk](#) so that `line_talker()` breaks out of the `while` loop as soon as it has printed the first string received (or simply remove the `while` loop). Once out of the `while` loop, the existing code calls `s.close()`.

(a). Start up the modified server, and connect to it with a client. Send a single message line, and use `netstat` to examine the TCP states of the client and server. What are these states?

(b). Send two message lines to the server. What are the TCP states of the client and server?

(c). Send three message lines to the server. Is there an error message at the client?

(d). Send two message lines to the server, while monitoring packets with [WireShark](#). The WireShark filter expression `tcp.port == 5431` may be useful for eliminating irrelevant traffic. What FIN packets do you see? Do you see a RST packet?

15.0. Outline a scenario in which TCP endpoint A sends data to B and then calls `close()` on its socket, and after the connection terminates B has not received all the data, even though the network has not failed. In the style of [12.6.1 The TCP Client](#), A's code might look like this:

```
s = new Socket(dest, destport);
sout = s.getOutputStream();
sout.write(large_buffer)
s.close()
```

Hint: see [12.7.2 Calling close\(\)](#).

