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 ⟨host,port⟩ 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. 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 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 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, S will also create a new thread or process to handle communication with s_C. Typically the server S will have multiple connected children of 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.

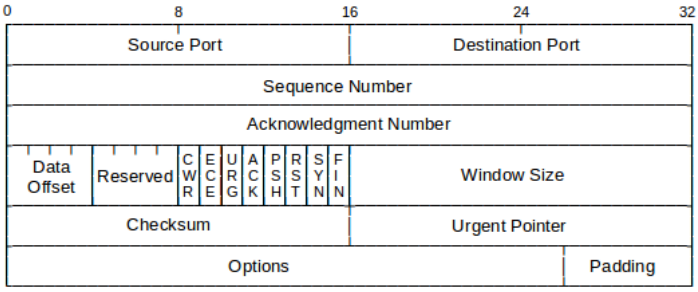
Saltzer, Reed and Clark categorized functions that were subject to the End-to-End principle this way:

> 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 checksum prevents delivery of corrupted data. The Data Offset is for specifying the number of words of Options.



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**.

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.

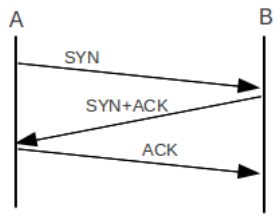
The TCP header defines six important flag bits; the brief definitions here are expanded upon in the sequel:

- **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, *14.8.2 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

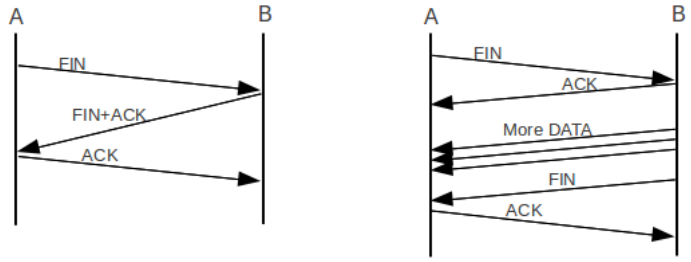
Normally, the three-way handshake is triggered by an application's request to connect; data can be sent only after the handshake completes. This means a one-RTT delay before any data can be sent. The original TCP standard RFC 793 does allow data to be sent with the first SYN packet, as part of the handshake, 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 earlier data in a more useful manner, so as to achieve request/reply turnaround comparable to that with RPC (*11.7 Remote Procedure Call (RPC)*). We return to this in *12.11 TCP Faster Opening*.

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

- 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



Two TCP close scenarios

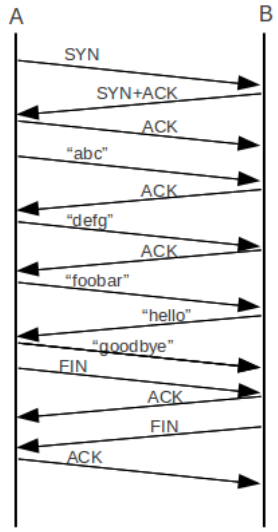
The FIN handshake is really more like two separate two-way FIN/ACK handshakes. If B is ready to close immediately, it may send its FIN along with its ACK of A's FIN, as is shown in the above diagram at the left. In theory this is rare, however, as the ACK of A's FIN is generated by the kernel but B's FIN cannot be sent until B's process is scheduled to run on the CPU. On the other hand, it is possible for B to send a considerable amount of data back to A after A sends its FIN, as is shown at the right. The FIN is, in effect, a promise not to *send* any more, but that side of the connection must still be prepared to receive data. 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`.

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	**seq=21**, ack=6, FIN	seq=6, **ack=21** ;; ACK of "goodbye"
12		seq=6, **ack=22** ;; ACK of FIN
13		seq=6, **ack=22**, FIN
14	**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	SYN, **seq=1000**	
2		SYN+ACK, seq=7000, **ack=1001**
3	ACK, **seq=1001**, ack=7001	
4	"abc", **seq=1001**, ack=7001	
5		ACK, seq=7001, **ack=1004**
6	"defg", **seq=1004**, ack=7001	
7		seq=7001, **ack=1008**
8	"foobar", **seq=1008**, ack=7001	
9		seq=7001, **ack=1014**, "hello"
10	**seq=1014**, ack=7006, "goodbye"	

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, either side can abort the connection at any time by sending RST.

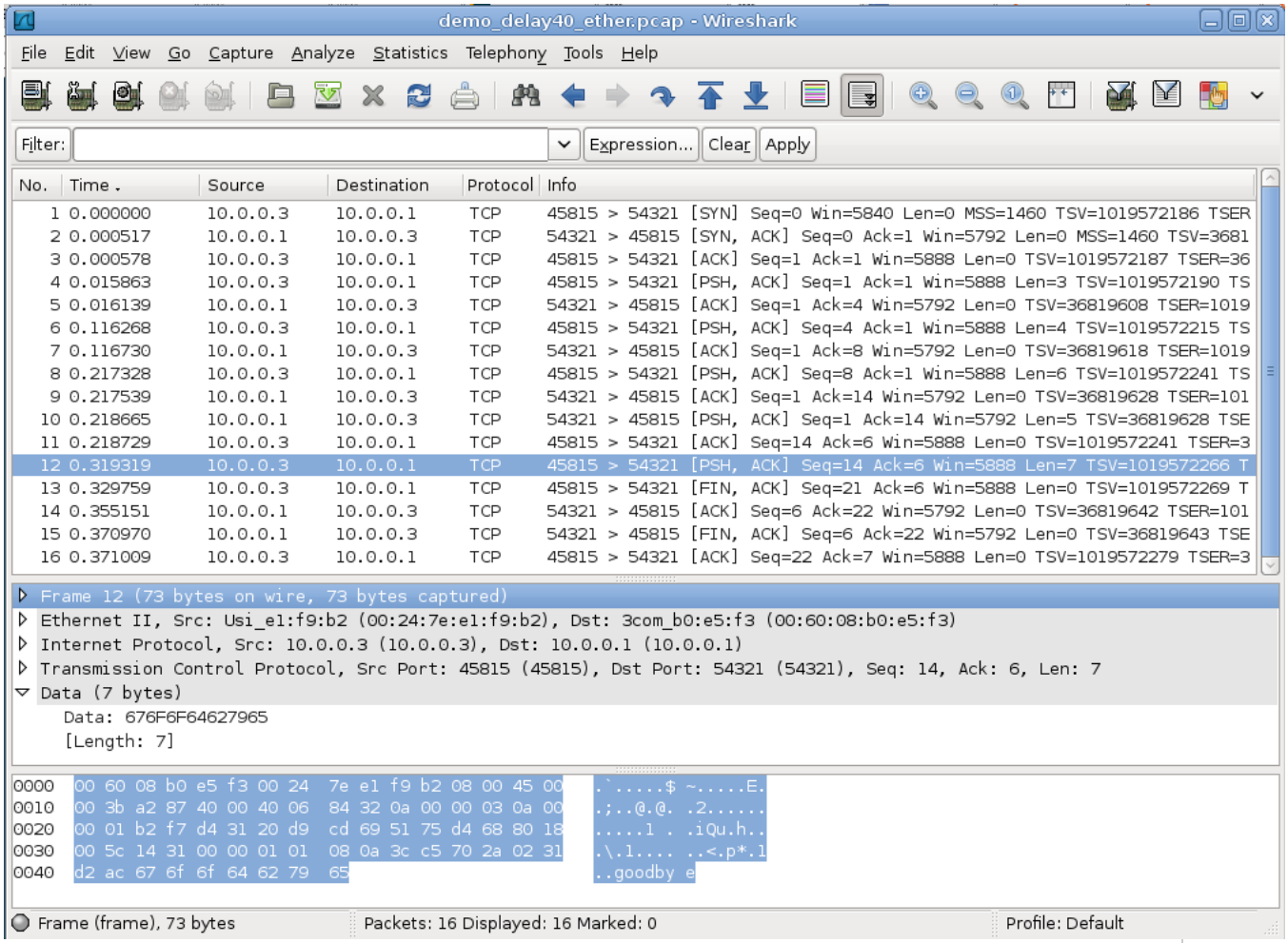
If A sends a series of small packets to B, then B has the option of assembling them into a full-sized I/O buffer before releasing them to the receiving application. However, if A sets the **PSH** bit on each packet, then B should release each packet immediately to the receiving application. In Berkeley 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 an ssh connection, in which A has sent a large amount of data to B, which is momentarily stalled processing it. The user at A wishes to abort the connection by sending the interrupt character CNTL-C. Under normal processing, 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 far ahead in the normal data stream. The receiver then skips processing the arriving data stream in first-come-first-served order and processes the urgent data first. For this to work, the receiving process must have signed up to receive an asynchronous signal when urgent data arrives.

The urgent data does appear in the ordinary TCP data stream, and it is up to the protocol to determine the length of the urgent data substring, and what to do with the unread, buffered data sent ahead of the urgent data. For the CNTL-C example, the urgent data consists of a single character, and the earlier data is simply discarded.

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 ⟨10.0.0.3,45815⟩ and side B (the server) uses ⟨10.0.0.1,54321⟩. WireShark is displaying relative 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 determined 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). 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.

Packets 11–14 in the table and 13–16 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 here was 551144795 and B's ISN was 1366676578. The actual pcap packet-capture file is at demo_tcp_connection.pcap. The hardware involved used **TCP checksum offloading** to have the network-interface card do the actual checksum calculations; as a result, the checksums are wrong in the actual capture file. WireShark has an option to disable the reporting of this.

12.5 TCP simplex-talk

Here is a Java version of the simplex-talk server for TCP. The main `while` loop has the call `ss.accept()` at the start; `ss` is the `ServerSocket` object and is in LISTEN state. This call blocks until an incoming connection is established, at which point it returns the connected child socket. Connections will be accepted from *all* IP addresses of the server host, *eg* the "normal" IP address and also the loopback address 127.0.0.1. Unlike the UDP case (*11.1.1.2 UDP and IP addresses*), the server response packets will always be sent from the same server IP address that the client first used to reach the server.

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`). In the former mode, if a second client connection is made while the first is connected, 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. The main loop is within the `line_talker()` call, and does not return to `ss.accept()` until the connection has closed.

In the latter 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
}
```

Here is the 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.

```
// 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
    }
}
```

Here are some things to try with THREADING=false:

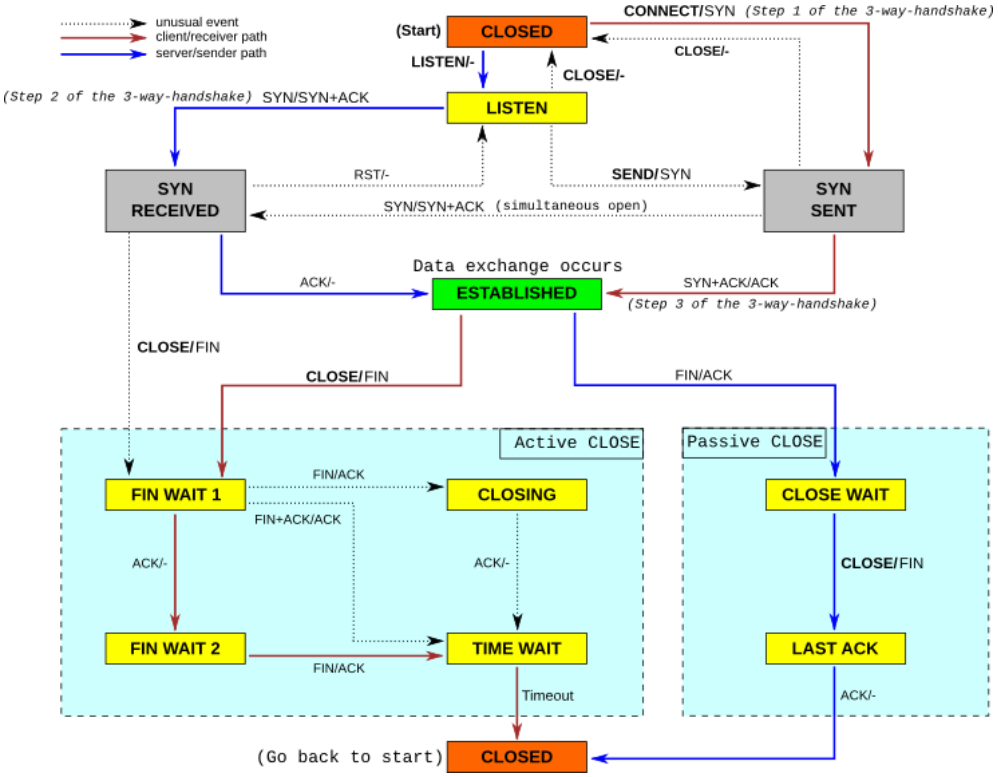
- start up two clients while the server is running. Type some lines into both. Then exit the first client.
- start up the client before the server.
- start up the server, and then the client. Type some text to the client. Kill the server. What happens to the client? (It may take a couple lines of input)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

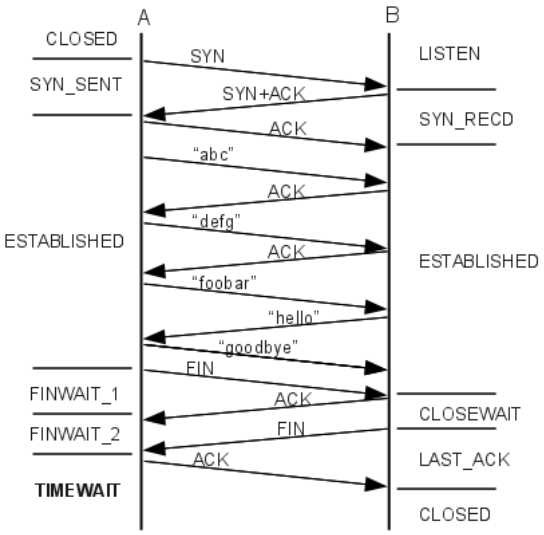
12.6 TCP state diagram

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. 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_RECD upon receipt of a SYN packet; the action is to respond with SYN+ACK.

In general, the state-diagram approach to protocol design has proven very effective, and is used for most protocols.



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



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 opening and closing connections.

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 takes the **Active CLOSE** path; the other side takes the **Passive CLOSE** path.

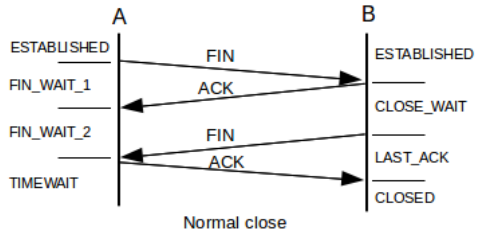
Although it essentially never occurs in practice, it is possible for each side to send the other a SYN, requesting a connection, **simultaneously** (that is, the SYNs cross on the wire). The telephony analogue occurs when each party dials the other simultaneously. On traditional land-lines, each party 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.

A simultaneous close – having both sides simultaneously send each other FINs – is a little more likely, though still not very. Each side would move to state FIN_WAIT_1. Then, upon receiving each other's FIN packets, each side would move to CLOSING, and then to TIMEWAIT.

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. A TCP endpoint is **half-open** if it is in the ESTABLISHED state, but in the meantime the other side has rebooted. As soon as the ESTABLISHED side sends a packet, the other side will respond with RST and the connection will be fully closed.

The "normal" close is



Normal close

In this scenario, A has moved from ESTABLISHED to FIN_WAIT_1 to FIN_WAIT_2 to TIMEWAIT (below) to CLOSED. B moves from ESTABLISHED to CLOSE_WAIT to LAST_ACK to CLOSED. All this essentially amounts to two separate two-way closure handshakes.

However, it is possible for B's ACK and FIN to be combined. In this case, A moves directly from FIN_WAIT_1 to TIMEWAIT. In order for this to happen, when A's FIN arrives at B, the socket-owning process at B must immediately wake up, recognize that A has closed its end, and immediately close its own end as well. This generates B's FIN; all this must happen before B's TCP layer sends the ACK of A's FIN. If the TCP layer adopts a policy of *immediately* sending ACKs upon receipt of any packet, this will never happen, as the FIN will arrive well before B's process can be scheduled to do anything. However, if B *delays* its ACKs slightly, then it is possible for B's ACK and FIN to be sent together.

Although this is not evident from the state diagram, 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 ones own current ACK. This includes the case where the receiver *acknowledges data not yet sent*.

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.

12.7 TCP Old Duplicates

Conceptually, perhaps the most serious threat facing the integrity of TCP data is external old duplicates (*11.2 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) ignored.

As with TFTP, coming up with a possible scenario for 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. T/TCP (*12.11 TCP Faster Opening*) used 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 *12.10 Anomalous TCP scenarios*.

12.8 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×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 (*11.2 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.

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 ⟨IP_address,port⟩ 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, 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.

12.9 The Three-Way Handshake Revisited

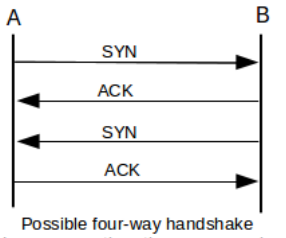
As stated earlier in *12.3 TCP Connection Establishment*, both sides choose an ISN; actual sequence numbers are the sum of the sender's ISN and the relative sequence number. There are two original reasons for this mechanism, and one later one (*12.9.1 ISNs and spoofing*). The original TCP specification, 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 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.



Possible four-way handshake
Less secure than three-way version

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.

12.9.1 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_A. 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_A with a considerable degree of accuracy. So M sends SYN to A with B as source, A sends SYN-ACK to B containing ISN_A, and M *guesses* this value and sends ACK(ISN_A+1) to A, again with B listed in the IP header as source, followed by a single-packet command.

This "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:

ISN = C(t) + hash(local_addr, local_port, remote_addr, remote_port, 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.

12.10 Anomalous TCP scenarios

TCP, like any transport protocol, must address the transport issues in *11.2 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. In modern TCP implementations this is addressed by 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. This is normally easy to arrange. An old duplicate due to sequence-number wraparound can now be rejected as having an old timestamp.

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.

12.11 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 *12.9.1 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 that 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 Internet Draft draft-ietf-tcpm-fastopen-05.txt, 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.

12.12 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 strategy is to send an initial data packet with the IP 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.

12.13 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".

12.14 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.

12.15 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

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) "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

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 backlogged "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.

common in our network." Today, it is unlikely that extra small packets would cause significant problems.

The Nagle algorithm does not always interact well with delayed ACKs, or with user expectations. It can usually be disabled on a per-connection basis. See also exercise 10.

12.16 TCP Flow Control

A TCP receiver may reduce the Window Size value of an open connection, thus informing the sender to switch to a smaller window size. This has nothing to do with congestion management but is instead related to **flow control**. This reduction appears in the ACKs sent back by the receiver. A given ACK is not supposed to reduce the window size by more than the size of the data it is acknowledging; in other words, the upper end of the window, lastACKed+winsize, is never supposed to get smaller. This means that no data sent by the sender will ever be retroactively invalidated by becoming beyond the upper edge of the window. A window might shrink from [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 window size to 0, this means that the sender may not send data. This would be done by the receiver to acknowledge that, yes, the data was received, but that more may not yet be sent. This corresponds to the ACK_{WAIT} suggested in *6.1.3 Flow Control*. Eventually, when the receiver is ready to receive data, it will send an ACK increasing the 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.

12.17 TCP Timeout and Retransmission

For TCP to work well for both intra-server-room and trans-global connections, the timeout value must *adapt*. TCP manages this by maintaining a running estimate of the RTT, EstRTT. In the original version, TCP then set TimeOut = 2×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 α, $0<\alpha<1$. Typical values of α might be $\alpha=1/2$ or $\alpha=7/8$. For α close to 1 this is very conservative in that EstRTT is slow to change. For α close to 0, EstRTT is very 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×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 β, $0<\beta<1$. Timeout was then set to EstRTT + 4×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.

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.

12.18 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

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.

12.19 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×MSL **TIMEWAIT**: a per-connection timer
- **Persist**: the timer used to poll the receiving end when winsize = 0
- **KeepAlive**, above

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".

12.20 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. That will be the focus of the next three chapters.

12.21 Exercises

1. 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?

2. 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.

3. 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 ⟨A,port1⟩ arrives at ⟨B,port2⟩, how would we tell to which of the two possible connections it belongs?

4. 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? Which node goes into TIMEWAIT state?

5. (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. Suppose that, after downloading a file, a user workstation is unplugged from the network. The workstation may or may not have first sent a FIN to start closing the connection.

(a). If the receiver has *not* sent the first FIN, what TCP states could this leave the server stuck in? Use the TCP state diagram.
(b). Now suppose the receiver *has* sent its FIN before being unplugged. What states could this leave the server stuck in?

7. 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. Suppose you are watching a video on a site like YouTube, where there is a progress bar showing how much of the video has been downloaded (and which hopefully stays comfortably head of the second progress bar showing your position in viewing the video). Occasionally, the download-progress bar jumps ahead by a modest but significant amount, instantaneously fast (much faster than the bandwidth alone could account for). At the TCP layer, what is going on to cause this jump? Hint: what does a TCP receiver perceive when a packet is lost and retransmitted?

9. 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 100 KB; the server should pause in sending when necessary to enforce this. An ordinary TCP connection will simply transfer data as fast as possible. What support, if any, do you need the receiving (client) application layer to provide, in order to enable this? What must the server-side application then do?

10. 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

11. 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 HANDOFF) 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/HANDOFF, 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.