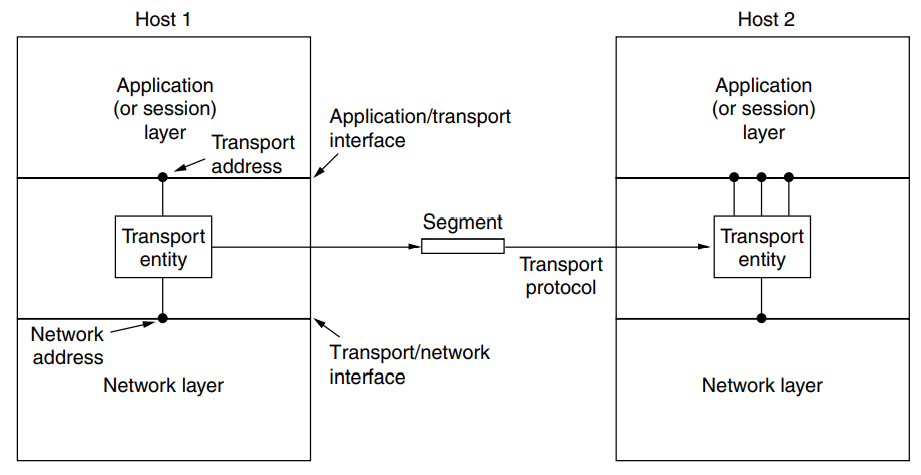
**CAPITULO 6**

Together with the network layer, the transport layer is the heart of the protocol hierarchy. The transport layer builds on the network layer to provide data transport from a process on a source machine to a process on a destination machine with a desired level of reliability that is independent of the physical networks currently in use.

**6.1 The Transport Service**

**6.1.1 Services Provided to the upper layers**

The ultimate goal of the transport layer is to provide efficient, reliable, and cost-effective data transmission service to its users, normally processes in the application layer. To achieve this, the transport layer makes use of the services provided by the network layer. The software and/or hardware within the transport layer that does the work is called the **transport entity**.



There are also two types of transport service. The **connection-oriented transport service** is similar to the connection-oriented network service in many ways. In both cases, connections have three phases: establishment, data transfer, and release.

However, note that it can be difficult to provide a connectionless transport service on top of a connection-oriented network service, since it is inefficient to set up a connection to send a single packet and then tear it down immediately afterwards.

If the transport layer service is so similar to the network layer service, why are there two distinct layers?

The transport code runs entirely on the users’ machines, but the network layer mostly runs on the routers, which are operated by the carrier (at least for a wide area network). The users have no real control over the network layer, so they cannot solve the problem of poor service by using better routers or putting more error handling in the data link layer because they don’t own the routers.

In essence, the existence of the transport layer makes it possible for the transport service to be more reliable than the underlying network. Hiding the network service behind a set of transport service primitives ensures that changing the network merely requires replacing one set of library procedures with another one that does the same thing with a different underlying service.

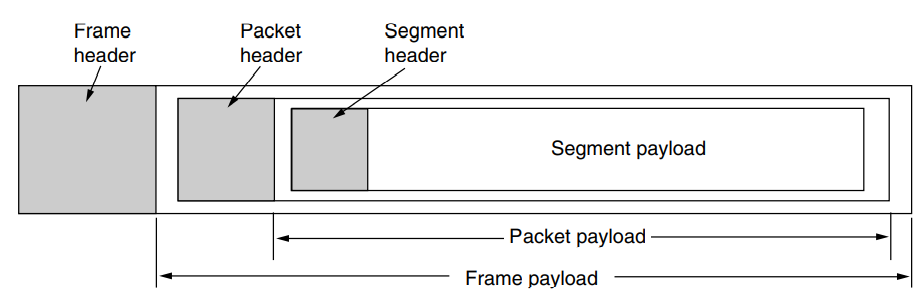
If all real networks were flawless and all had the same service primitives and were guaranteed never, ever to change, the transport layer might not be needed. However, in the real world it fulfils the key function of isolating the upper layers from the technology, design, and imperfections of the network.

The bottom four layers can be seen as the **transport service provider**, whereas the upper layer(s) are the **transport service user**. This distinction of provider versus user has a considerable impact on the design of the layers and puts the transport layer in a key position, since it forms the major boundary between the provider and user of the reliable data transmission service. It is the level that applications see.

**6.1.2 Services Provided to the upper layers**

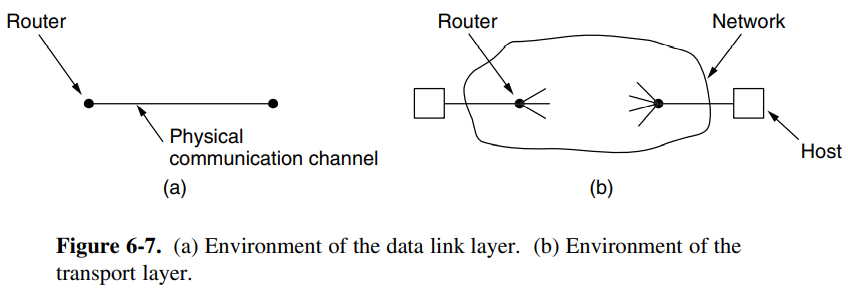
For lack of a better term, we will use the term **segment** for messages sent from transport entity to transport entity. TCP, UDP and other Internet protocols use this term. Some older protocols used the ungainly name **TPDU (Transport Protocol Data Unit)**.

Thus, segments (exchanged by the transport layer) are contained in packets (exchanged by the network layer). In turn, these packets are contained in frames (exchanged by the data link layer).



**6.2 Elements of transport protocols**

The transport service is implemented by a **transport protocol** used between the two transport entities. In some ways, transport protocols resemble the data link protocols. Both have to deal with error control, sequencing, and flow control, among other issues.

However, significant differences between the two also exist. These differences are due to major dissimilarities between the environments in which the two protocols operate.

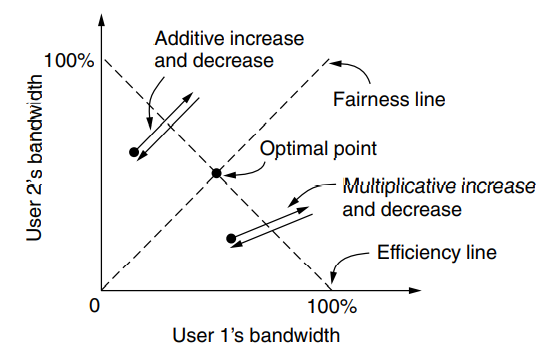
At the data link layer, two routers communicate directly via a physical channel, whether wired or wireless, whereas at the transport layer, this physical channel is replaced by the entire network. This difference has many important implications for the protocols.

**6.3 Congestion Control**

**6.3.2 Regulating the Sending rate**

**AIMD (Additive Increase Multiplicative Decrease)** is the appropriate control law to arrive at the efficient and fair operating point.

The graph below shows the bandwidth allocated to user 1 on the x-axis and to user 2 on the y-axis. When the allocation is fair, both users will receive the same amount of bandwidth. This is shown by the dotted fairness line.

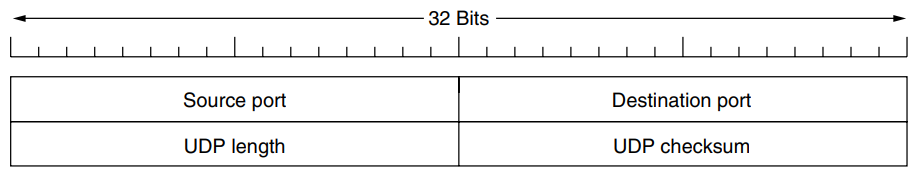
When the allocations sum to 100%, the capacity of the link, the allocation is efficient. This is shown by the dotted efficiency line. A congestion signal is given by the network to both users when the sum of their allocations crosses this line. The intersection of these lines is the desired operating point, when both users have the same bandwidth, and all of the network bandwidth is used.

**6.4 THE INTERNET TRANSPORT PROTOCOLS: UDP**

The Internet has two main protocols in the transport layer, a connectionless protocol and a connection-oriented one. The protocols complement each other. The connectionless protocol is UDP. It does almost nothing beyond sending packets between applications, letting applications build their own protocols on top as needed. The connection-oriented protocol is TCP. It does almost everything. It makes connections and adds reliability with retransmissions, along with flow control and congestion control, all on behalf of the applications that use it.

**6.4.1 Introduction to UDP**

The Internet protocol suite supports a connectionless transport protocol called **UDP (User Datagram Protocol)**.

UDP transmits segments consisting of an 8-byte header followed by the payload. The two ports serve to identify the endpoints within the source and destination machines. When a UDP packet arrives, its payload is handed to the process attached to the destination port.

Think of ports as mailboxes that applications can rent to receive packets.

The source port is primarily needed when a reply must be sent back to the source. By copying the *Source port* field from the incoming segment into the *Destination port* field of the outgoing segment, the process sending the reply can specify which process on the sending machine is to get it.

The *UDP length field* includes the 8-byte header and the data. The minimum length is 8 bytes, to cover the header. The maximum length is 65,515 bytes, which is lower than the largest number that will fit in 16 bits because of the size limit on IP packets.

An optional *Checksum* is also provided for extra reliability. It checksums the header, the data, and a conceptual IP pseudoheader.

**6.5 THE INTERNET TRANSPORT PROTOCOLS: TCP**

**6.5.1 Introduction to TCP**

**TCP (Transmission Control Protocol)** was specifically designed to provide a reliable end-to-end byte stream over an unreliable internetwork. An internetwork differs from a single network because different parts may have wildly different topologies, bandwidths, delays, packet sizes, and other parameters. TCP was designed to dynamically adapt to properties of the internetwork and to be robust in the face of many kinds of failures.

Each machine supporting TCP has a TCP transport entity, either a library procedure, a user process, or most commonly part of the kernel. In all cases, it manages TCP streams and interfaces to the IP layer.

The IP layer gives no guarantee that datagrams will be delivered properly, nor any indication of how fast datagrams may be sent. It is up to TCP to send datagrams fast enough to make use of the capacity but not cause congestion, and to time out and retransmit any datagrams that are not delivered.

Datagrams that do arrive may well do so in the wrong order; it is also up to TCP to reassemble them into messages in the proper sequence. In short, TCP must furnish good performance with the reliability that most applications want and that IP does not provide.

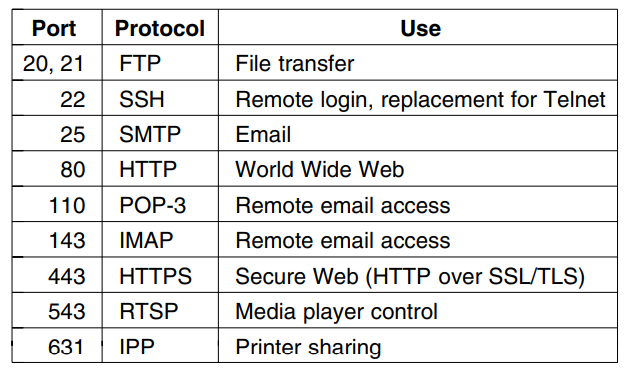
**6.5.2 The TCP service model**

TCP service is obtained by both the sender and the receiver creating end points, called **sockets**. Each socket has a socket number (address) consisting of the IP address of the host and a 16-bit number local to that host, called a **port**.

A port is the TCP name for a TSAP. For TCP service to be obtained, a connection must be explicitly established between a socket on one machine and a socket on another machine.

A socket may be used for multiple connections at the same time. Connections are identified by the socket identifiers at both ends, that is, (socket1, socket2). No virtual circuit numbers or other identifiers are used.

Port numbers below 1024 are reserved for standard services that can usually only be started by privileged users. They are called **well-known ports**.



All TCP connections are full duplex and point-to-point. Full duplex means that traffic can go in both directions at the same time. Point-to-point means that each connection has exactly two end points. TCP does not support multicasting or broadcasting.

A TCP connection is a byte stream, not a message stream. There is no way for the receiver to detect the unit(s) in which the data were written, no matter how hard it tries.

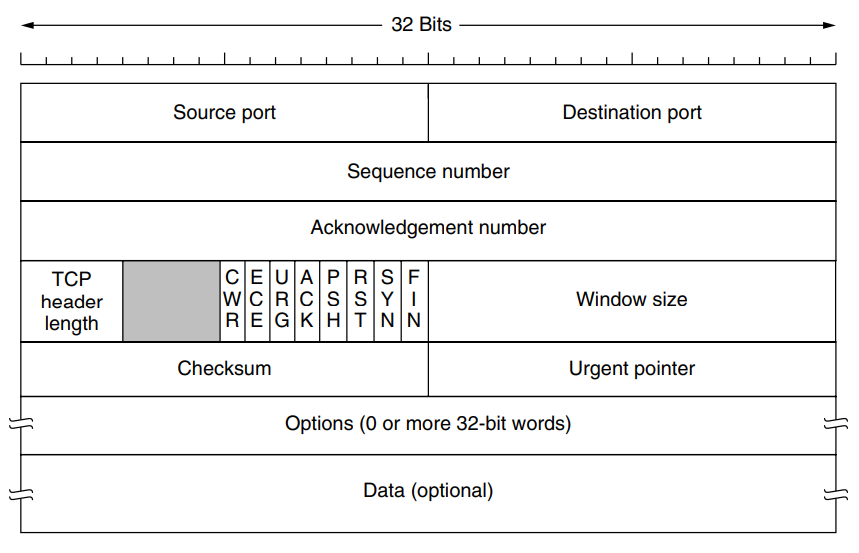
**6.5.3 The TCP Protocol**

A key feature of TCP, and one that dominates the protocol design, is that every byte on a TCP connection has its own 32-bit sequence number.

The sending and receiving TCP entities exchange data in the form of segments. A TCP segment consists of a fixed 20-byte header (plus an optional part) followed by zero or more data bytes. The TCP software decides how big segments should be.

**6.5.4 The TCP Segment Header**

Every segment begins with a fixed-format, 20-byte header. The fixed header may be followed by header options. After the options, if any, up to 65,535 − 20 − 20 = 65,495 data bytes may follow, where the first 20 refer to the IP header and the second to the TCP header. Segments without any data are legal and are commonly used for acknowledgements and control messages.



The *Source port* and *Destination port* fields identify the local end points of the connection. A TCP port plus its host’s IP address forms a 48-bit unique end point. The source and destination end points together identify the connection. This connection identifier is called a **5 tuple** because it consists of five pieces of information: the protocol (TCP), source IP and source port, and destination IP and destination port.

The *Sequence number* and *Acknowledgement number* fields perform their usual functions. Note that the latter specifies the next in-order byte expected, not the last byte correctly received. It is a **cumulative acknowledgement** because it summarizes the received data with a single number. Both are 32 bits because every byte of data is numbered in a TCP stream.

The TCP *header length* tells how many 32-bit words are contained in the TCP header. This information is needed because the *Options* field is of variable length, so the header is, too.

Flow control in TCP is handled using a variable-sized sliding window. The *Window size* field tells how many bytes may be sent starting at the byte acknowledged. A *Window size field* of 0 is legal and says that the bytes up to and including *Acknowledgement number* − 1 have been received, but that the receiver has not had a chance to consume the data and would like no more data for the moment, thank you. The receiver can later grant permission to send by transmitting a segment with the same *Acknowledgement number* and a nonzero *Window size* field.

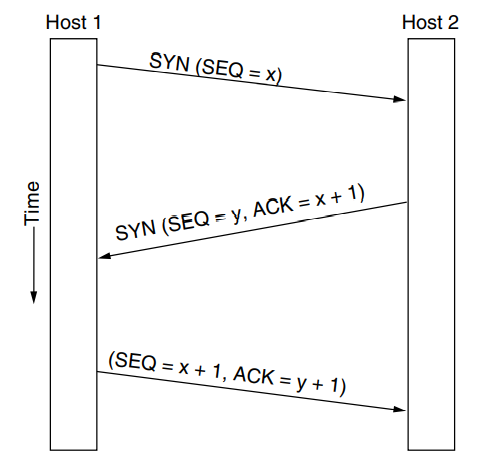
A *Checksum* is also provided for extra reliability. It checksums the header, the data, and a conceptual pseudoheader in exactly the same way as UDP.

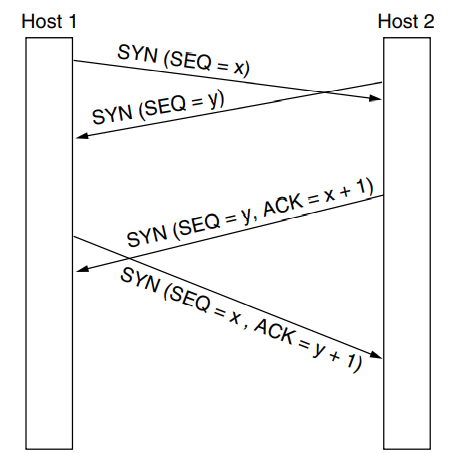
**6.5.5 TCP Connection Establishment**

Connections are established in TCP by means of the three-way handshake. To establish a connection, one side, say, the server, passively waits for an incoming connection by executing the LISTEN and ACCEPT primitives in that order, either specifying a specific source or nobody in particular.

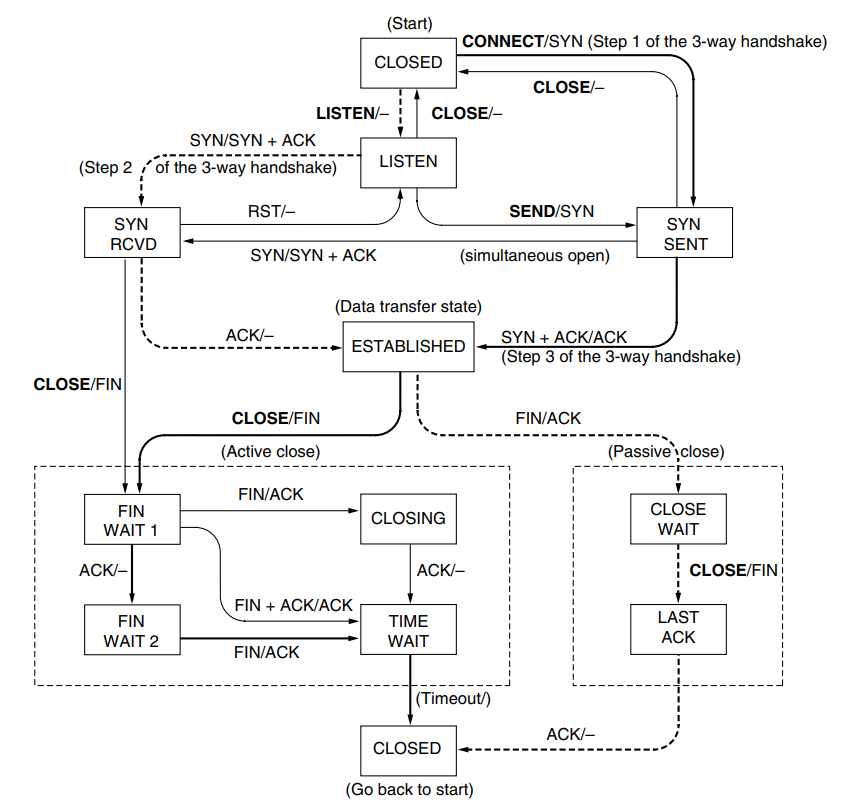
The other side, say, the client, executes a CONNECT primitive, specifying the IP address and port to which it wants to connect, the maximum TCP segment size it is willing to accept, and optionally some user data. The CONNECT primitive sends a TCP segment with the SYN bit on and ACK bit off and waits for a response.

When this segment arrives at the destination, the TCP entity there checks to see if there is a process that has done a LISTEN on the port given in the Destination port field. If not, it sends a reply with the RST bit on to reject the connection.

If some process is listening to the port, that process is given the incoming TCP segment. It can either accept or reject the connection. If it accepts, an acknowledgement segment is sent back.

In the event that two hosts simultaneously attempt to establish a connection between the same two sockets, the result is that just one connection is established, not two, because connections are identified by their end points.

**6.5.7 TCP Connection Management Modelling**

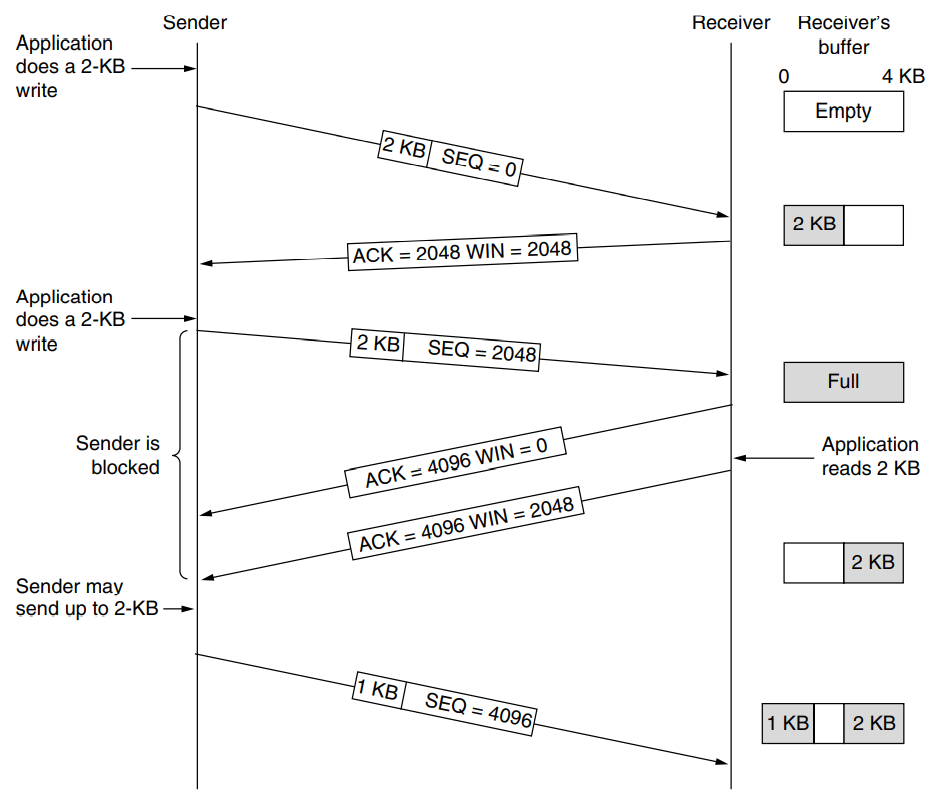


TCP connection management finite state machine. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. Each transition is labelled with the event causing it and the action resulting from it, separated by a slash.

**6.5.8 TCP Sliding Window**

Window management in TCP decouples the issues of acknowledgement of the correct receipt of segments and receiver buffer allocation.

Suppose the receiver has a 4096-byte buffer. If the sender transmits a 2048-byte segment that is correctly received, the receiver will acknowledge the segment. However, since it now has only 2048 bytes of buffer space (until the application removes some data from the buffer), it will advertise a window of 2048 starting at the next byte expected.

Now the sender transmits another 2048 bytes, which are acknowledged, but the advertised window is of size 0. The sender must stop until the application process on the receiving host has removed some data from the buffer, at which time TCP can advertise a larger window and more data can be sent.

When the window is 0, the sender may not normally send segments, with two exceptions. First, urgent data may be sent, for example, to allow the user to kill the process running on the remote machine. Second, the sender may send a 1-byte segment to force the receiver to reannounce the next byte expected and the window size. This packet is called a **window probe**. The TCP standard explicitly provides this option to prevent deadlock if a window update ever gets lost.

Senders are not required to transmit data as soon as they come in from the application. Neither are receivers required to send acknowledgements as soon as possible.

**6.5.9 TCP Timer Management**

TCP uses multiple timers (at least conceptually) to do its work. The most important of these is the **RTO (Retransmission TimeOut)**. When a segment is sent, a retransmission timer is started. If the segment is acknowledged before the timer expires, the timer is stopped. If, on the other hand, the timer goes off before the acknowledgement comes in, the segment is retransmitted.

The question that arises is: how long should the timeout be?

The solution is to use a dynamic algorithm that constantly adapts the timeout interval, based on continuous measurements of network performance.

For each connection, TCP maintains a variable, *SRTT (Smoothed Round-Trip Time)*, that is the best current estimate of the round-trip time to the destination in question. When a segment is sent, a timer is started, both to see how long the acknowledgement takes and also to trigger a retransmission if it takes too long.

If the acknowledgement gets back before the timer expires, TCP measures how long the acknowledgement took, say, R. It then updates SRTT according to the formula:



Where α is a smoothing factor that determines how quickly the old values are forgotten. Typically, α = 7/8. This kind of formula is an **EWMA (Exponentially Weighted Moving Average)** or low-pass filter that discards noise in the samples.

Experience soon showed that a constant value was too inflexible because it failed to respond when the variance went up. In particular, queueing models of random traffic predict that when the load approaches capacity, the delay becomes large and highly variable.

This can lead to the retransmission timer firing and a copy of the packet being retransmitted although the original packet is still transiting the network. It is all the more likely to happen under conditions of high load, which is the worst time at which to send additional packets into the network.

To fix this problem, the timeout was made value sensitive to the variance in round-trip times as well as the smoothed round-trip time. This change requires keeping track of another smoothed variable, *RTTVAR (RoundTrip Time VARiation)* that is updated using the formula



This is an EWMA as before, and typically β = 3/4. The retransmission timeout, *RTO*, is set to be:

The choice of the factor 4 is somewhat arbitrary, but multiplication by 4 can be done with a single shift, and less than 1% of all packets come in more than four standard deviations late.

One problem that occurs with gathering the samples, R, of the round-trip time is what to do when a segment times out and is sent again. When the acknowledgement comes in, it is unclear whether the acknowledgement refers to the first transmission or a later one. Guessing wrong can seriously contaminate the retransmission timeout.

Phil Kar made a simple proposal: do not update estimates on any segments that have been retransmitted. Additionally, the timeout is doubled on each successive retransmission until the segments get through the first time. This fix is called **Karn’s algorithm**. Most TCP implementations use it.

**6.5.10 TCP Congestion Control**

When the load offered to any network is more than it can handle, congestion builds up. The Internet is no exception. The network layer detects congestion when queues grow large at routers and tries to manage it, if only by dropping packets.

It is up to the transport layer to receive congestion feedback from the network layer and slow down the rate of traffic that it is sending into the network.

A transport protocol using an AIMD (Additive Increase Multiplicative Decrease) control law in response to binary congestion signals from the network would converge to a fair and efficient bandwidth allocation.

TCP congestion control is based on implementing this approach using a window and with packet loss as the binary signal. To do so, TCP maintains a **congestion window** whose size is the number of bytes the sender may have in the network at any time. The corresponding rate is the window size divided by the round-trip time of the connection. TCP adjusts the size of the window according to the AIMD rule.

Recall that the congestion window is maintained in addition to the flow control window, which specifies the number of bytes that the receiver can buffer. Both windows are tracked in parallel, and the number of bytes that may be sent is the smaller of the two windows. Thus, the effective window is the smaller of what the sender thinks is all right and what the receiver thinks is all right. TCP will stop sending data if either the congestion or the flow control window is temporarily full.

The high-level fix implemented was to approximate an AIMD congestion window. The interesting part, and much of the complexity of TCP congestion control, is how this was added to an existing implementation without changing any of the message formats, which made it instantly deployable.

To start, it was observed that packet loss is a suitable signal of congestion. This signal comes a little late (as the network is already congested) but it is quite dependable. After all, it is difficult to build a router that does not drop packets when it is overloaded.

However, using packet loss as a congestion signal depends on transmission errors being relatively rare. This is not normally the case for wireless links, which is why they include their own retransmission mechanism at the link layer.

Because of wireless retransmissions, network layer packet loss due to transmission errors is normally masked on wireless networks. It is also rare on other links because wires and optical fibers typically have low bit-error rates.

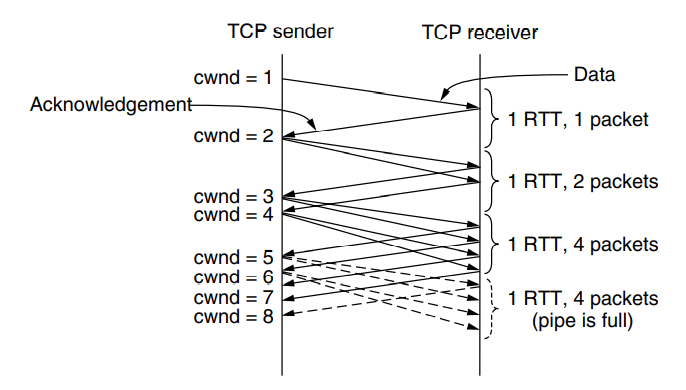
A second consideration is that the AIMD rule will take a very long time to reach a good operating point on fast networks if the congestion window is started from a small size.

When a connection is established, the sender initializes the congestion window to a small initial value of at most four segments; the use of four segments is an increase from an earlier initial value of one segment based on experience. The sender then sends the initial window. The packets will take a round-trip time to be acknowledged. For each segment that is acknowledged before the retransmission timer goes off, the sender adds one segment’s worth of bytes to the congestion window. Plus, as that segment has been acknowledged, there is now one less segment in the network. The upshot is that every acknowledged segment allows two more segments to be sent. The congestion window is doubling every roundtrip time.

This algorithm is called **slow start**, but it is not slow at all except in comparison to the previous algorithm that let an entire flow control window be sent all at once.

In the first round-trip time, the sender injects one packet into the network (and the receiver receives one packet). Two packets are sent in the next round-trip time, then four packets in the third round-trip time.

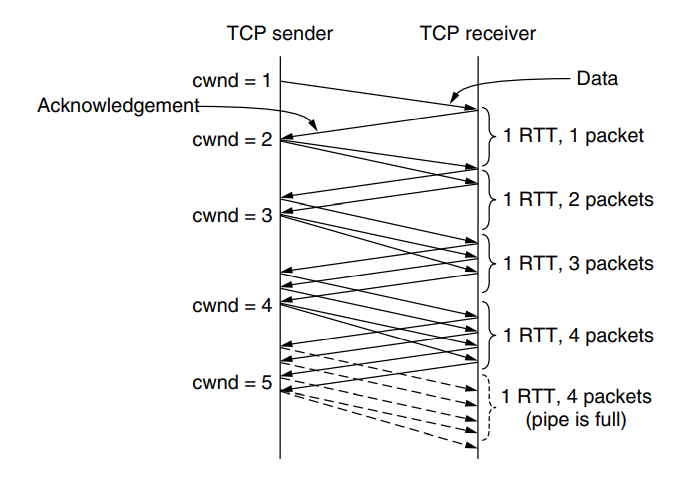
Slow-start works well over a range of link speeds and round-trip times, and uses an ack clock to match the rate of sender transmissions to the network path. When the sender gets an acknowledgement, it increases the congestion window by one and immediately sends two packets into the network.



Because slow start causes exponential growth, eventually it will send too many packets into the network too quickly. When this happens, queues will build up in the network. When the queues are full, one or more packets will be lost.

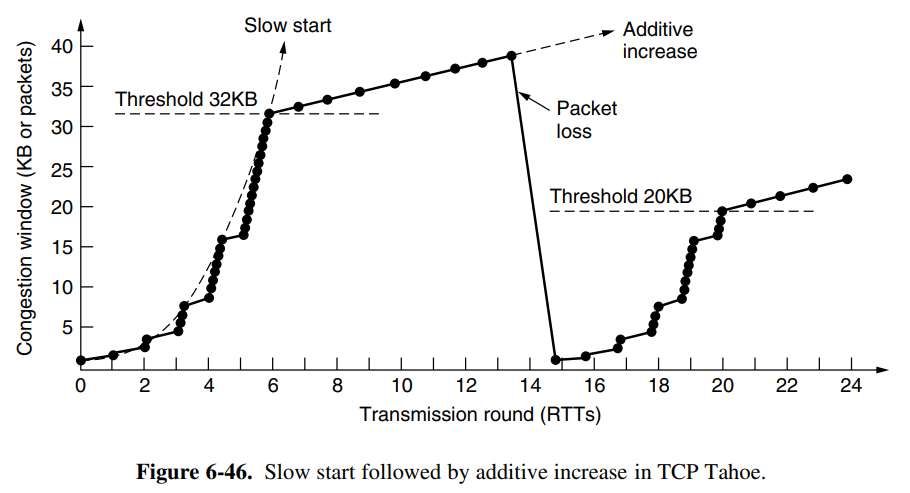
To keep slow start under control, the sender keeps a threshold for the connection called the **slow start threshold**. Initially this value is set arbitrarily high, to the size of the flow control window, so that it will not limit the connection. TCP keeps increasing the congestion window in slow start until a timeout occurs or the congestion window exceeds the threshold (or the receiver’s window is filled).

Whenever a packet loss is detected, for example, by a timeout, the slow start threshold is set to be half of the congestion window and the entire process is restarted. The idea is that the current window is too large because it caused congestion previously that is only now detected by a timeout.

Whenever the slow start threshold is crossed, TCP switches from slow start to additive increase. In this mode, the congestion window is increased by one segment every round-trip time.

At the end of every RTT, the sender’s congestion window has grown enough that it can inject an additional packet into the network. Compared to slow start, the linear rate of growth is much slower.

There is a quick way for the sender to recognize that one of its packets has been lost. As packets beyond the lost packet arrive at the receiver, they trigger acknowledgements that return to the sender. These acknowledgements bear the same acknowledgement number. They are called **duplicate acknowledgements**. Each time the sender receives a duplicate acknowledgement, it is likely that another packet has arrived at the receiver and the lost packet still has not shown up.

Thus, TCP somewhat arbitrarily assumes that three duplicate acknowledgements imply that a packet has been lost. The identity of the lost packet can be inferred from the acknowledgement number as well. This heuristic is called **fast retransmission**. After it fires, the slow start threshold is still set to half the current congestion window, just as with a timeout.

One large change has also affected TCP implementations.

Much of the complexity of TCP comes from inferring from a stream of duplicate acknowledgements which packets have arrived and which packets have been lost. The cumulative acknowledgement number does not provide this information. A simple fix is the use of **SACK (Selective ACKnowledgements)**, which lists up to three ranges of bytes that have been received. With this information, the sender can more directly decide what packets to retransmit and track the packets in flight to implement the congestion window.

When the sender and receiver set up a connection, they each send the *SACK permitted* TCP option to signal that they understand selective acknowledgements. A receiver uses the TCP *Acknowledgement number* field in the normal manner, as a cumulative acknowledgement of the highest in-order byte that has been received. When it receives packet 3 out of order (because packet 2 was lost), it sends a *SACK option* for the received data along with the (duplicate) cumulative acknowledgement for packet 1. The *SACK option* gives the byte ranges that have been received above the number given by the cumulative acknowledgement. The first range is the packet that triggered the duplicate acknowledgement.

