TCP Performance

July 2000

Geoff Huston

The *Transmission Control Protocol* (TCP) and the User Datagram Protocol (UDP) are both IP transport-layer protocols. UDP is a lightweight protocol that allows applications to make direct use of the unreliable datagram service provided by the underlying IP service. UDP is commonly used to support applications that use simple query/response transactions, or applications that support real-time communications. TCP provides a reliable data-transfer service, and is used for both bulk data transfer and interactive data applications. TCP is the major transport protocol in use in most IP networks, and supports the transfer of over 90 percent of all traffic across the public Internet today. Given this major role for TCP, the performance of this protocol forms a significant part of the total picture of service performance for IP networks. In this article we examine TCP in further detail, looking at what makes a TCP session perform reliably and well. This article draws on material published in the *Internet Performance Survival Guide* [1].

Overview of TCP

TCP is the embodiment of reliable end-to-end transmission functionality in the overall Internet architecture. All the functionality required to take a simple base of IP datagram delivery and build upon this a control model that implements reliability, sequencing, flow control, and data streaming is embedded within TCP [2].

TCP provides a communication channel between processes on each host system. The channel is reliable, full-duplex, and streaming. To achieve this functionality, the TCP drivers break up the session data stream into discrete segments, and attach a TCP header to each segment. An IP header is attached to this TCP packet, and the composite packet is then passed to the network for delivery. This TCP header has numerous fields that are used to support the intended TCP functionality. TCP has the following functional characteristics:

- *Unicast protocol* : TCP is based on a unicast network model, and supports data exchange between precisely two parties. It does not support broadcast or multicast network models.
- *Connection state* : Rather than impose a state within the network to support the connection, TCP uses synchronized state between the two endpoints. This synchronized state is set up as part of an initial connection process, so TCP can be regarded as a connection-oriented protocol. Much of the protocol design is intended to ensure that each local state transition is communicated to, and acknowledged by, the remote party.
- *Reliable* : Reliability implies that the stream of octets passed to the TCP driver at one end of the connection will be transmitted across the network so that the stream is presented to the remote process as the same sequence of octets, in the same order as that generated by the sender.
  This implies that the protocol detects when segments of the data stream have been discarded by the network, reordered, duplicated, or corrupted. Where necessary, the

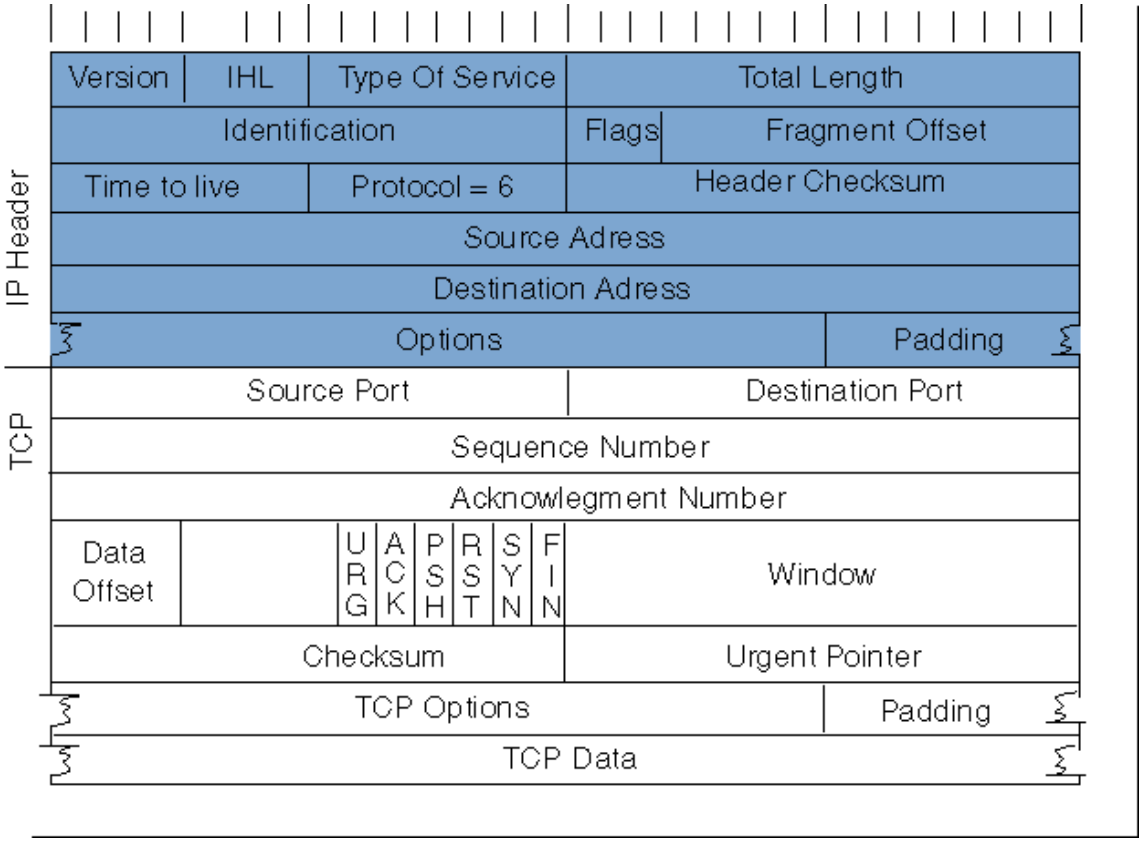
sender will retransmit damaged segments so as to allow the receiver to reconstruct the original data stream. This implies that a TCP sender must maintain a local copy of all transmitted data until it receives an indication that the receiver has completed an accurate transfer of the data.

- *Full duplex* : TCP is a full-duplex protocol; it allows both parties to send and receive data within the context of the single TCP connection.
- *Streaming* : Although TCP uses a packet structure for network transmission, TCP is a true streaming protocol, and application-level network operations are not transparent. Some protocols explicitly encapsulate each application transaction; for every *write* , there must be a matching *read* . In this manner, the application-derived segmentation of the data stream into a logical record structure is preserved across the network. TCP does not preserve such an implicit structure imposed on the data stream, so that there is no pairing between *write* and *read* operations within the network protocol. For example, a TCP application may *write* three data blocks in sequence into the network connection, which may be collected by the remote reader in a single *read* operation. The size of the data blocks (segments) used in a TCP session is negotiated at the start of the session. The sender attempts to use the largest segment size it can for the data transfer, within the constraints of the maximum segment size of the receiver, the maximum segment size of the configured sender, and the maxi-mum supportable non-fragmented packet size of the network path (path *Maximum Transmission Unit* [MTU]). The path MTU is refreshed periodically to adjust to any changes that may occur within the network while the TCP connection is active.
- *Rate adaptation* : TCP is also a rate-adaptive protocol, in that the rate of data transfer is intended to adapt to the prevailing load conditions within the network and adapt to the processing capacity of the receiver. There is no predetermined TCP data-transfer rate; if the network and the receiver both have additional available capacity, a TCP sender will attempt to inject more data into the network to take up this available space. Conversely, if there is congestion, a TCP sender will reduce its sending rate to allow the network to recover. This adaptation function attempts to achieve the highest possible data-transfer rate without triggering consistent data loss.

The TCP Protocal Header

The TCP header structure, shown in Figure 1, uses a pair of 16-bit source and destination Port addresses. The next field is a 32-bit sequence number, which identifies the sequence number of the first data octet in this packet. The sequence number does not start at an initial value of 1 for each new TCP connection; the selection of an initial value is critical, because the initial value is intended to prevent delayed data from an old connection from being incorrectly interpreted as being valid within a current connection. The sequence number is necessary to ensure that arriving packets can be ordered in the sender?s original order. This field is also used within the flow-control structure to allow the association of a data packet with its corresponding acknowledgement, allowing a sender to estimate the current round-trip time across the network.

Figure 1: The TCP/IP Datagram



The *acknowledgment sequence number* is used to inform the remote end of the data that has been successfully received. The acknowledgment sequence number is actually one greater than that of the last octet correctly received at the local end of the connection. The *data offset* field indicates the number of four-octet words within the TCP header. Six single *bit flags* are used to indicate various conditions. URG is used to indicate whether the urgent pointer is valid. ACK is used to indicate whether the *acknowledgment* field is valid. PSH is set when the sender wants the remote application to push this data to the remote application. RST is used to reset the connection. SYN (for *synchronize*) is used within the connection startup phase, and FIN (for *finish*) is used to close the connection in an orderly fashion. The *window* field is a 16-bit count of available buffer space. It is added to the acknowledgment sequence number to indicate the highest sequence number the receiver can accept. The TCP *checksum* is applied to a synthesized header that includes the source and destination addresses from the outer IP datagram. The final field in the TCP header is the urgent pointer, which, when added to the sequence number, indicates the sequence number of the final octet of urgent data if the urgent flag is set.

Many options can be carried in a TCP header. Those relevant to TCP performance include:

- *Maximum-receive-segment-size option* : This option is used when the connection is being opened. It is intended to inform the remote end of the maximum segment size, measured in octets, that the sender is willing to receive on the TCP connection. This option is used only in the initial SYN packet (the initial packet exchange that opens a TCP connection). It sets both the maximum receive segment size and the maximum size of the advertised TCP window, passed to the remote end of the connection. In a robust implementation of TCP, this option should be used with path MTU discovery to establish a

segment size that can be passed across the connection without fragmentation, an essential attribute of a high-performance data flow.

- *Window-scale option* : This option is intended to address the issue of the maximum window size in the face of paths that exhibit a high-delay bandwidth product. This option allows the window size advertisement to be right-shifted by the amount specified (in binary arithmetic, a right-shift corresponds to a multiplication by 2). Without this option, the maximum window size that can be advertised is 65,535 bytes (the maximum value obtainable in a 16-bit field). The limit of TCP transfer speed is effectively one window size in transit between the sender and the receiver. For high-speed, long-delay networks, this performance limitation is a significant factor, because it limits the transfer rate to at most 65,535 bytes per round-trip interval, regardless of available network capacity. Use of the window-scale option allows the TCP sender to effectively adapt to high-band-width, high-delay network paths, by allowing more data to be held in flight. The maximum window size with this option is 2^{30} bytes. This option is negotiated at the start of the TCP connection, and can be sent in a packet only with the SYN flag. Note that while an MTU discovery process allows optimal setting of the maximum-receive-segment-size option, no corresponding bandwidth delay product discovery allows the reliable automated setting of the window-scale option [3].
- SACK-*permitted option and* SACK *option* : This option alters the acknowledgment behavior of TCP. SACK is an acronym for *selective acknowledgment* . The SACK-permitted option is offered to the remote end during TCP setup as an option to an opening SYN packet. The SACK option permits selective acknowledgment of permitted data. The default TCP acknowledgment behavior is to acknowledge the highest sequence number of in-order bytes. This default behavior is prone to cause unnecessary retransmission of data, which can exacerbate a congestion condition that may have been the cause of the original packet loss. The SACK option allows the receiver to modify the acknowledgment field to describe noncontinuous blocks of received data, so that the sender can retransmit only what is missing at the receiver's end [4].

Any robust high-performance implementation of TCP should negotiate these parameters at the start of the TCP session, ensuring the following: that the session is using the largest possible IP packet size that can be carried without fragmentation, that the window sizes used in the transfer are adequate for the bandwidth-delay product of the network path, and that selective acknowledgment can be used for rapid recovery from line-error conditions or from short periods of marginally degraded network performance.

TCP Operation

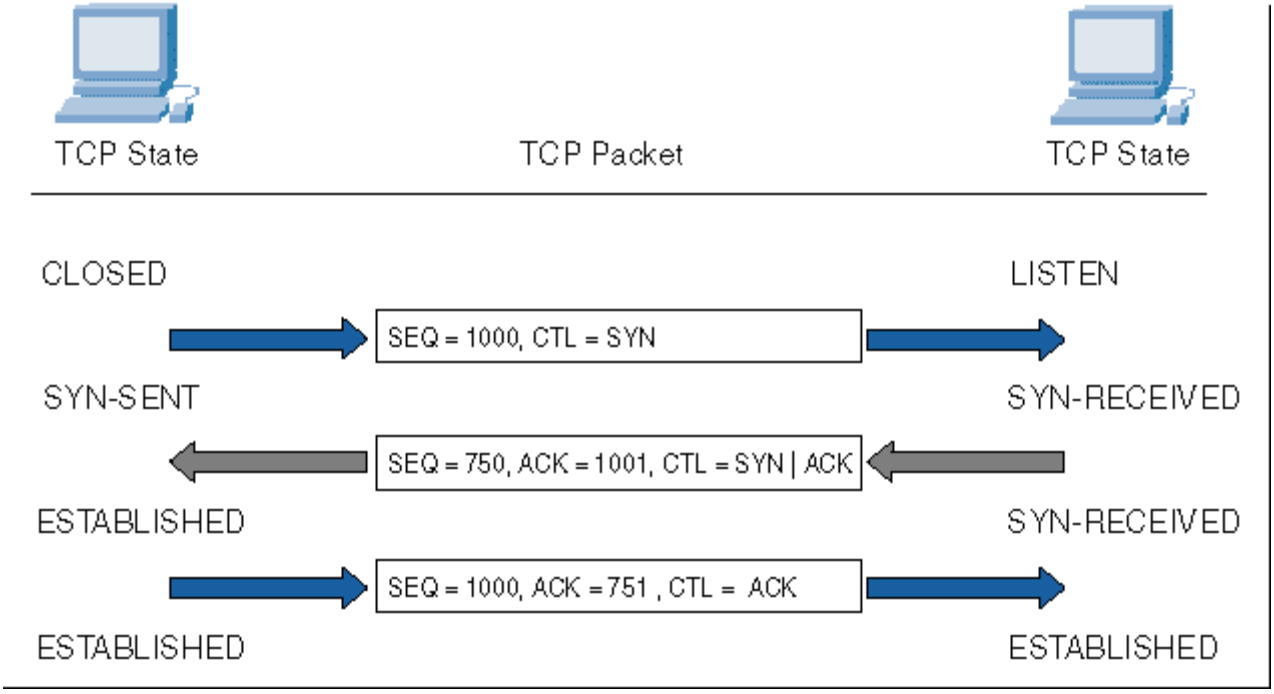
The first phase of a TCP session is establishment of the connection. This requires a three-way handshake, ensuring that both sides of the connection have an unambiguous understanding of the sequence number space of the remote side for this session. The operation of the connection is as follows:

- The local system sends the remote end an initial sequence number to the remote port, using a SYN packet.
- The remote system responds with an ACK of the initial sequence number and the initial sequence number of the remote end in a response SYN packet.
- The local end responds with an ACK of this remote sequence number.

The connection is opened.

The operation of this algorithm is shown in Figure 2. The performance implication of this protocol exchange is that it takes one and a half *round-trip times* (RTTs) for the two systems to synchronize state before any data can be sent.

Figure 2 : TCP Connection Handshake



After the connection has been established, the TCP protocol manages the reliable exchange of data between the two systems. The algorithms that determine the various retransmission timers have been redefined numerous times. TCP is a sliding-window protocol, and the general principle of flow control is based on the management of the advertised window size and the management of retransmission timeouts, attempting to optimize protocol performance within the observed delay and loss parameters of the connection. Tuning a TCP protocol stack for optimal performance over a very low-delay, high-bandwidth LAN requires different settings to obtain optimal performance over a dialup Internet connection, which in turn is different for the requirements of a high-speed wide-area network. Although TCP attempts to discover the delay bandwidth product of the connection, and attempts to automatically optimize its flow rates within the estimated parameters of the network path, some estimates will not be accurate, and the corresponding efforts by TCP to optimize behavior may not be completely successful.

Another critical aspect is that TCP is an adaptive flow-control protocol. TCP uses a basic flow-control algorithm of increasing the data-flow rate until the network signals that some form of saturation level has been reached (normally indicated by data loss). When the sender receives an indication of data loss, the TCP flow rate is reduced; when reliable transmission is reestablished, the flow rate slowly increases again.

If no reliable flow is reestablished, the flow rate backs further off to an initial probe of a single packet, and the entire adaptive flow-control process starts again.

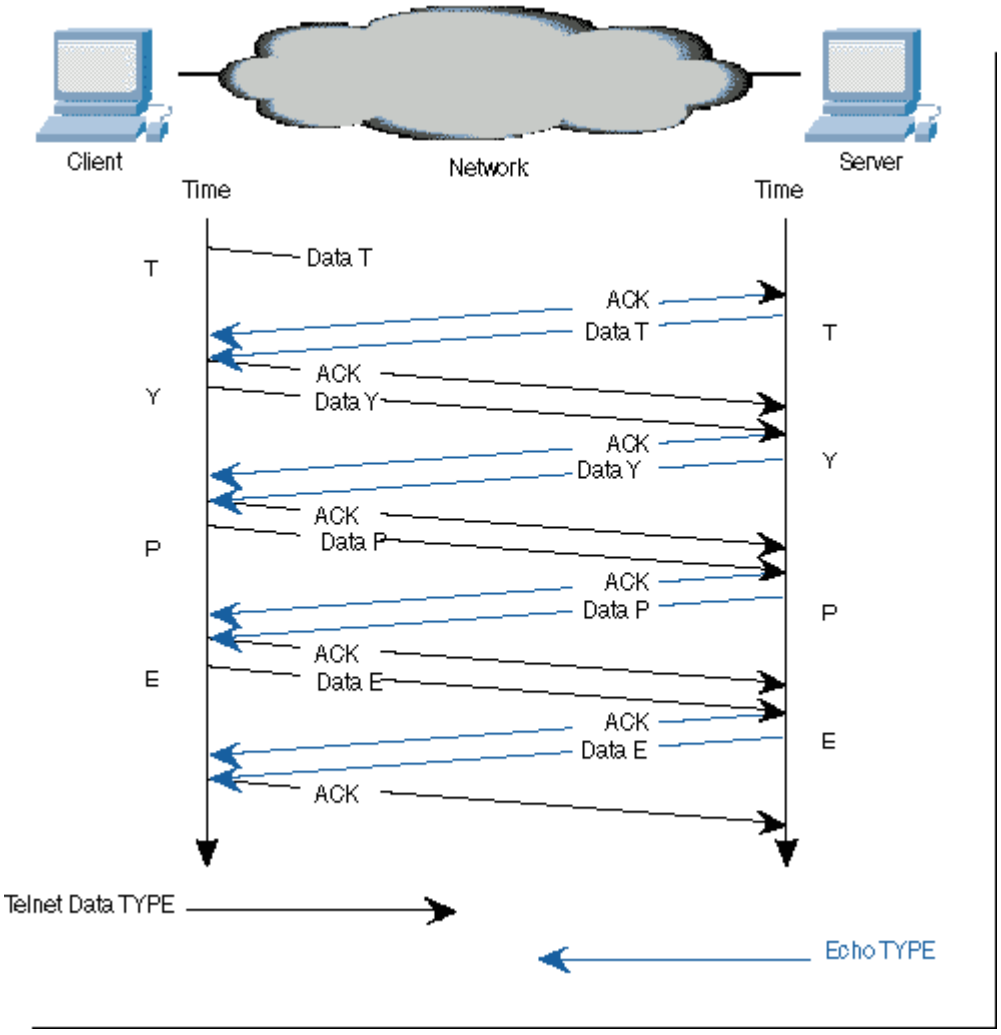
This process has numerous results relevant to service quality. First, TCP behaves *adaptively* , rather than *predictively* . The flow-control algorithms are intended to increase the data-flow rate to fill all available network path capacity, but they are also intended to quickly back off if the available capacity changes because of interaction with other traffic, or if a dynamic change occurs in the end-to-end network path. For example, a single TCP flow across an otherwise idle network attempts to fill the network path with data, optimizing the flow rate within the available network capacity. If a second TCP flow opens up across the same path, the two flow-control algorithms will interact so that both flows will stabilize to use approximately half of the available capacity per flow. The objective of the TCP algorithms is to adapt so that the network is fully used whenever one or more data flows are present. In design, tension always exists between

the efficiency of network use and the enforcement of predictable session performance. With TCP, you give up predictable throughput but gain a highly utilized, efficient network.

Interactive TCP

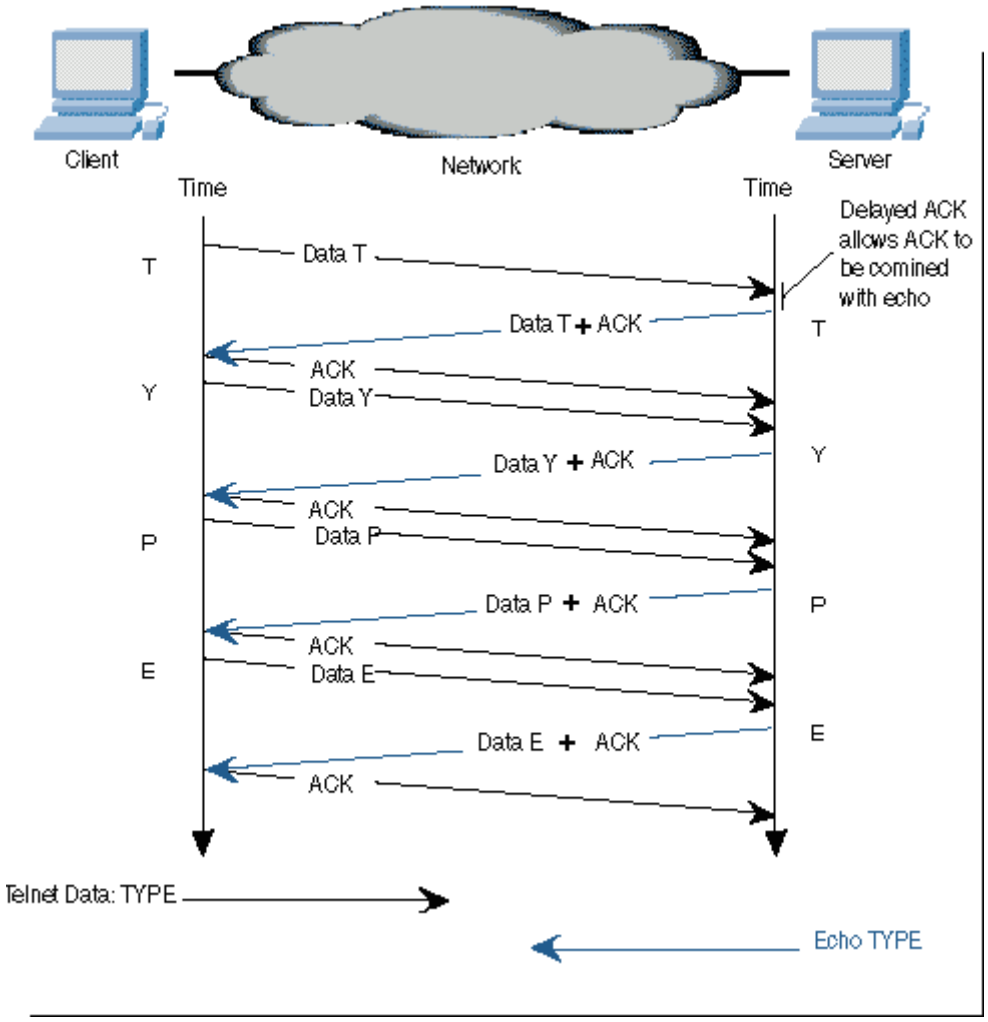
Interactive protocols are typically directed at supporting single character interactions, where each character is carried in a single packet, as is its echo. The protocol interaction to support this is indicated in Figure 3.

Figure 3: Interactive Exchange



These 2 bytes of data generate four TCP/IP packets, or 160 bytes of protocol overhead. TCP makes some small improvement in this exchange through the use of *piggybacking* , where an ACK is carried in the same packet as the data, and *delayed acknowledgment* , where an ACK is delayed up to 200 ms before sending, to give the server application the opportunity to generate data that the ACK can piggyback. The resultant protocol exchange is indicated in Figure 4.

Figure 4: Intereactive Exchange with Delayed ACK



For short-delay LANs, this protocol exchange offers acceptable performance. This protocol exchange for a single data character and its echo occurs within about 16 ms on an Ethernet LAN, corresponding to an interactive rate of 60 characters per second. When the network delay is increased in a WAN, these small packets can be a source of congestion load. The TCP mechanism to address this small-packet congestion was described by John Nagle in RFC 896 [5]. Commonly referred to as the *Nagle Algorithm* , this mechanism inhibits a sender from transmitting any additional small segments while the TCP connection has outstanding unacknowledged small segments. On a LAN, this modification to the algorithm has a negligible effect; in contrast, on a WAN, it has a dramatic effect in reducing the number of small packets in direct correlation to the network path congestion level (as shown in Figures 5 and 6). The cost is an increase in session jitter by up to a round-trip time interval. Applications that are jitter-sensitive typically disable this control algorithm.

Figure 5: Wan Interactive Exchange

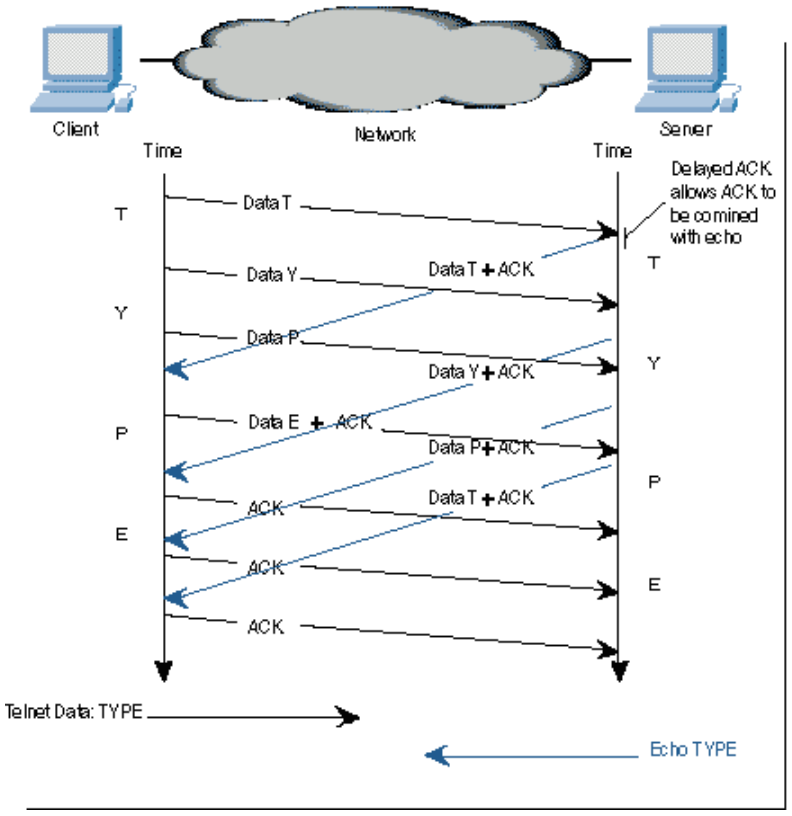
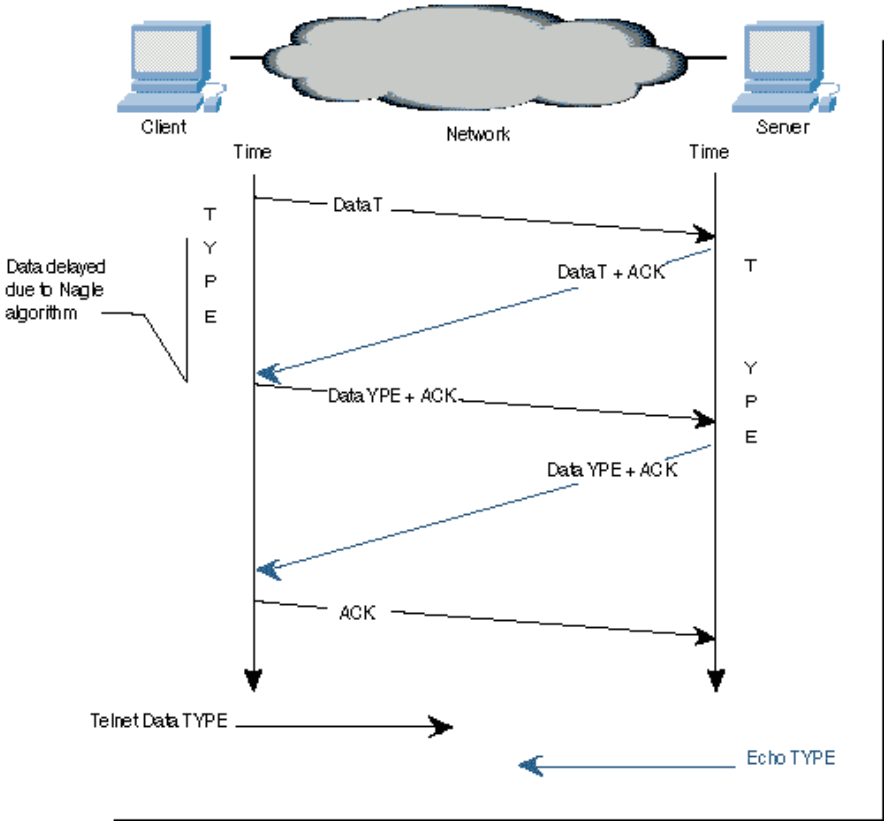


Figure 6: Wan Interactive Exchange with Nagle Algorithm



TCP is not a highly efficient protocol for the transmission of interactive traffic. The typical carriage efficiency of the protocol across a LAN is 2 bytes of payload and 120 bytes of protocol overhead. Across a WAN, the Nagle algorithm may improve this carriage efficiency slightly by increasing the number of bytes of payload for each payload transaction, although it will do so at the expense of increased session jitter.

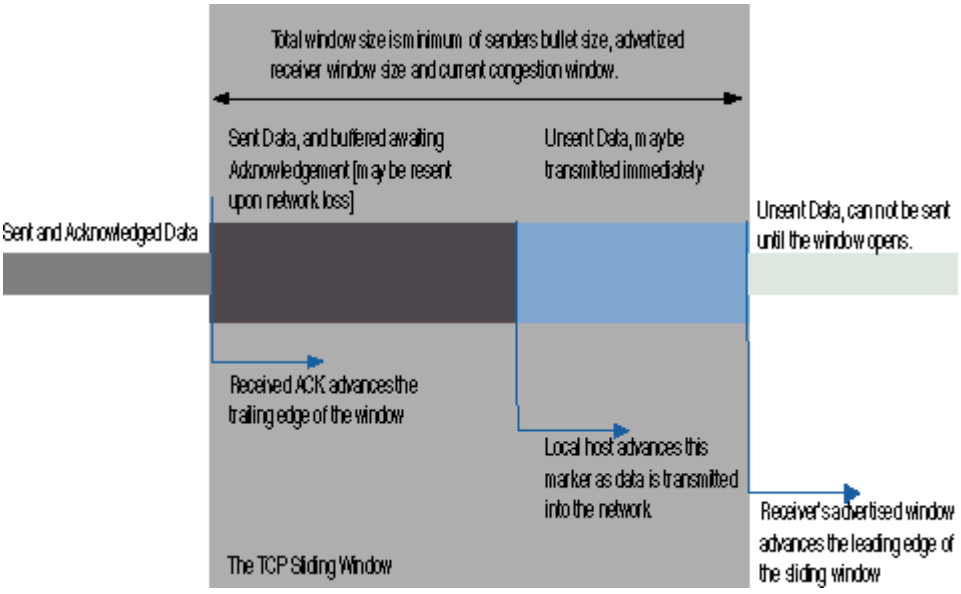
TCP Volume Transfer

The objective for this application is to maximize the efficiency of the data transfer, implying that TCP should endeavor to locate the point of dynamic equilibrium of maximum network efficiency, where the sending data rate is maximized just prior to the onset of sustained packet loss.

Further increasing the sending rate from such a point will run the risk of generating a congestion condition within the network, with rapidly increasing packet-loss levels. This, in turn, will force the TCP protocol to retransmit the lost data, resulting in reduced data-transfer efficiency. On the other hand, attempting to completely eliminate packet-loss rates implies that the sender must reduce the sending rate of data into the network so as not to create transient congestion conditions along the path to the receiver. Such an action will, in all probability, leave the network with idle capacity, resulting in inefficient use of available network resources.

The notion of a point of equilibrium is an important one. The objective of TCP is to coordinate the actions of the sender, the network, and the receiver so that the network path has sufficient data such that the network is not idle, but it is not so overloaded that a congestion backlog builds up and data loss occurs. Maintaining this point of equilibrium requires the sender and receiver to be synchronized so that the sender passes a packet into the network at precisely the same time as the receiver removes a packet from the network. If the sender attempts to exceed this equilibrium rate, network congestion will occur. If the sender attempts to reduce its rate, the efficiency of the network will drop. TCP uses a sliding-window protocol to support bulk data transfer (Figure 7).

Figure 7: TCP Sliding Window



The receiver advertises to the sender the available buffer space at the receiver. The sender can transmit up to this amount of data before having to await a further buffer update from the receiver. The sender should have no more than this amount of data in transit in the network. The sender must also buffer sent data until it has been ACKed by the receiver. The send window is the minimum of the sender's buffer size and the advertised receiver window. Each time an

ACK is received, the trailing edge of the send window is advanced. The minimum of the sender's buffer and the advertised receiver's window is used to calculate a new leading edge. If this send window encompasses unsent data, this data can be sent immediately.

The size of TCP buffers in each host is a critical limitation to performance in WANs. The protocol is capable of transferring one send window of data per round-trip interval. For example, with a send window of 4096 bytes and a transmission path with an RTT of 600 ms, a TCP session is capable of sustaining a maximum transfer rate of 48 Kbps, regardless of the bandwidth of the network path. Maximum efficiency of the transfer is obtained only if the sender is capable of completely filling the network path with data. Because the sender will have an amount of data in forward transit and an equivalent amount of data awaiting reception of an ACK signal, both the sender's buffer and the receiver's advertised window should be no smaller than the Delay-Bandwidth Product of the network path. That is:

Window size (le or eq) Bandwidth (bytes/sec) (times) Round-trip time (sec)

The 16-bit field within the TCP header can contain values up to 65,535, imposing an upper limit on the available window size of 65,535 bytes. This imposes an upper limit on TCP performance of some 64 KB per RTT, even when both end systems have arbitrarily large send and receive buffers. This limit can be modified by the use of a window-scale option, described in RFC 1323, effectively increasing the size of the window to a 30-bit field, but transmitting only the most significant 16 bits of the value. This allows the sender and receiver to use buffer sizes that can operate efficiently at speeds that encompass most of the current very-high-speed network transmission technologies across distances of the scale of the terrestrial intercontinental cable systems.

Although the maximum window size and the RTT together determine the maximum achievable data-transfer rate, there is an additional element of flow control required for TCP. If a TCP session commenced by injecting a full window of data into the network, then there is a strong probability that much of the initial burst of data would be lost because of transient congestion, particularly if a large window is being used. Instead, TCP adopts a more conservative approach by starting with a modest amount of data that has a high probability of successful transmission, and then probing the network with increasing amounts of data for as long as the network does not show signs of congestion. When congestion is experienced, the sending rate is dropped and the probing for additional capacity is resumed.

The dynamic operation of the window is a critical component of TCP performance for volume transfer. The mechanics of the protocol involve an additional overriding modifier of the sender's window, the *congestion window* , referred to as *cwnd* . The objective of the window-management algorithm is to start transmitting at a rate that has a very low probability of packet loss, then to increase the rate (by increasing the *cwnd* size) until the sender receives an indication, through the detection of packet loss, that the rate has exceeded the available capacity of the network. The sender then immediately halves its sending rate by reducing the value of *cwnd* , and resumes a gradual increase of the sending rate. The goal is to continually modify the sending rate such that it oscillates around the true value of available network capacity. This oscillation enables a dynamic adjustment that automatically senses any increase or decrease in available capacity through the lifetime of the data flow.

The intended outcome is that of a dynamically adjusting cooperative data flow, where a combination of such flows behaves fairly, in that each flow obtains essentially a fair share of the network, and so that close to maximal use of available network resources is made. This flow-control functionality is achieved through a combination of *cwnd* value management and packet-loss and retransmission algorithms. TCP flow control has three major parts: the flow-control modes of *Slow Start* and Congestion Avoidance, and the response to packet loss that determines how TCP switches between these two modes of operation.

TCP Slow Start

The starting value of the *cwnd* window (the Initial Window, or IW) is set to that of the Sender Maximum Segment Size (SMSS) value. This SMSS value is based on the receiver's maximum segment size, obtained during the SYN handshake, the discovered path MTU (if used), the MTU of the sending interface, or, in the absence of other information, 536 bytes. The sender then enters a flow-control mode termed *Slow Start* .

The sender sends a single data segment, and because the window is now full, it then awaits the corresponding ACK. When the ACK is received, the sender increases its window by increasing the value of *cwnd* by the value of SMSS. This then allows the sender to transmit two segments; at that point, the congestion window is again full, and the sender must await the corresponding ACKs for these segments. This algorithm continues by increasing the value of *cwnd* (and, correspondingly, opening the size of the congestion window) by one SMSS for every ACK received that acknowledges new data.

If the receiver is sending an ACK for every packet, the effect of this algorithm is that the data rate of the sender doubles every round-trip time interval. If the receiver supports delayed ACKs, the rate of increase will be slightly lower, but nevertheless the rate will increase by a minimum of one SMSS each round-trip time. Obviously, this cannot be sustained indefinitely. Either the value of *cwnd* will exceed the advertised receive window or the sender's window, or the capacity of the network will be exceeded, in which case packets will be lost.

There is another limit to the slow-start rate increase, maintained in a variable termed *ssthresh* , or *Slow-Start Threshold* . If the value of *cwnd* increases past the value of *ssthresh*, the TCP flow-control mode is changed from *Slow Start* to congestion avoidance. Initially the value of *ssthresh* is set to the receiver's maximum window size. However, when congestion is noted, *ssthresh* is set to half the current window size, providing TCP with a memory of the point where the onset of network congestion may be anticipated in future.

One aspect to highlight concerns the interaction of the slow-start algorithm with high-capacity long-delay networks, the so-called Long Fat Networks (or LFNs, pronounced "elephants"). The behavior of the slow-start algorithm is to send a single packet, await an ACK, then send two packets, and await the corresponding ACKs, and so on. The TCP activity on LFNs tends to cluster at each epoch of the round-trip time, with a quiet period that follows after the available window of data has been transmitted. The received ACKs arrive back at the sender with an inter-ACK spacing that is equivalent to the data rate of the bottleneck point on the network path. During *Slow Start* , the sender transmits at a rate equal to twice this bottleneck rate. The rate adaptation function that must occur within the network takes place in the router at the entrance to the bottleneck point. The sender's packets arrive at this router at twice the rate of egress from the router, and the router stores the overflow within its internal buffer. When this buffer overflows, packets will be dropped, and the slow-start phase is over. The important conclusion is that the sender will stop increasing its data rate when there is buffer exhaustion, a condition that may not be the same as reaching the true available data rate. If the router has a buffer capacity considerably less than the delay-bandwidth product of the egress circuit, the two values are certainly not the same.

In this case, the TCP slow-start algorithm will finish with a sending rate that is well below the actual available capacity. The efficient operation of TCP, particularly in LFNs, is critically reliant on adequately large buffers within the network routers.

Another aspect of *Slow Start* is the choice of a single segment as the initial sending window. Experimentation indicates that an initial value of up to four segments can allow for a more efficient session startup, particularly for those short-duration TCP sessions so prevalent with Web fetches [6]. Observation of Web traffic indicates an average Web data transfer of 17 segments. A *slow start* from one segment will take five RTT intervals to transfer this data, while

using an initial value of four will reduce the transfer time to three RTT intervals. However, four segments may be too many when using low-speed links with limited buffers, so a more robust approach is to use an initial value of no more than two segments to commence *Slow Start* [7].

Packet Loss

Slow Start attempts to start a TCP session at a rate the network can support and then continually increase the rate. How does TCP know when to stop this increase? This slow-start rate increase stops when the congestion window exceeds the receiver's advertised window, when the rate exceeds the remembered value of the onset of congestion as recorded in ssthresh, or when the rate is greater than the network can sustain. Addressing the last condition, how does a TCP sender know that it is sending at a rate greater than the network can sustain? The answer is that this is shown by data packets being dropped by the network. In this case, TCP has to undertake many functions:

- The packet loss has to be detected by the sender.
- The missing data has to be retransmitted.
- The sending data rate should be adjusted to reduce the probability of further packet loss.

TCP can detect packet loss in two ways. First, if a single packet is lost within a sequence of packets, the successful delivery packets following the lost packet will cause the receiver to generate a *duplicate* ACK for each successive packet The reception of these duplicate ACKs is a signal of such packet loss. Second, if a packet is lost at the end of a sequence of sent packets, there are no following packets to generate duplicate ACKs. In this case, there are no corresponding ACKs for this packet, and the sender's retransmit timer will expire and the sender will assume packet loss.

A single duplicate ACK is not a reliable signal of packet loss. When a TCP receiver gets a data packet with an out-of-order TCP sequence value, the receiver must generate an immediate ACK of the highest in-order data byte received. This will be a duplicate of an earlier transmitted ACK. Where a single packet is lost from a sequence of packets, all subsequent packets will generate a duplicate ACK packet.

On the other hand, where a packet is rerouted with an additional incremental delay, the reordering of the packet stream at the receiver's end will generate a small number of duplicate ACKs, followed by an ACK of the entire data sequence, after the errant packet is received. The sender distinguishes between these cases by using three duplicate ACK packets as a signal of packet loss.

The third duplicate ACK triggers the sender to immediately send the segment referenced by the duplicate ACK value (*fast retransmit*) and commence a sequence termed *Fast Recovery* . In fast recovery, the value of *ssthresh* is set to half the current send window size (the send window is the amount of unacknowledged data outstanding). The congestion window, *cwnd* , is set three segments greater than *ssthresh* to allow for three segments already buffered at the receiver. If this allows additional data to be sent, then this is done. Each additional duplicate ACK inflates *cwnd* by a further segment size, allowing more data to be sent. When an ACK arrives that encompasses new data, the value of*cwnd* is set back to ssthresh, and TCP enters congestion-avoidance mode. Fast Recovery is intended to rapidly repair single packet loss, allowing the sender to continue to maintain the ACK-clocked data rate for new data while the packet loss repair is being undertaken. This is because there is still a sequence of ACKs arriving at the sender, so that the network is continuing to pass timing signals to the sender indicating the rate at which packets are arriving at the receiver. Only when the repair has been completed does the sender drop its window to the *ssthresh* value as part of the transition to congestion-avoidance mode [8].

The other signal of packet loss is a complete cessation of any ACK packets arriving to the sender. The sender cannot wait indefinitely for a delayed ACK, but must make the assumption at some

point in time that the next unacknowledged data segment must be retransmitted. This is managed by the sender maintaining a *Retransmission Timer* . The maintenance of this timer has performance and efficiency implications. If the timer triggers too early, the sender will push duplicate data into the network unnecessarily. If the timer triggers too slowly, the sender will remain idle for too long, unnecessarily slowing down the flow of data. The TCP sender uses a timer to measure the elapsed time between sending a data segment and receiving the corresponding acknowledgment. Individual measurements of this time interval will exhibit significant variance, and implementations of TCP use a smoothing function when updating the retransmission timer of the flow with each measurement. The commonly used algorithm was originally described by Van Jacobson [9], modified so that the retransmission timer is set to the smoothed round-trip-time value, plus four times a smoothed mean deviation factor [10].

When the retransmission timer expires, the actions are similar to that of duplicate ACK packets, in that the sender must reduce its sending rate in response to congestion. The threshold value, *ssthresh* , is set to half of the current value of outstanding unacknowledged data, as in the duplicate ACK case. However, the sender cannot make any valid assumptions about the current state of the network, given that no useful information has been provided to the sender for more than one RTT interval. In this case, the sender closes the congestion window back to one segment, and restarts the flow in *slow start* -mode by sending a single segment. The difference from the initial *slow start* is that, in this case, the *ssthresh* value is set so that the sender will probe the congestion area more slowly using a linear sending rate increase when the congestion window reaches the remembered ssthresh value.

Congestion Avoidance

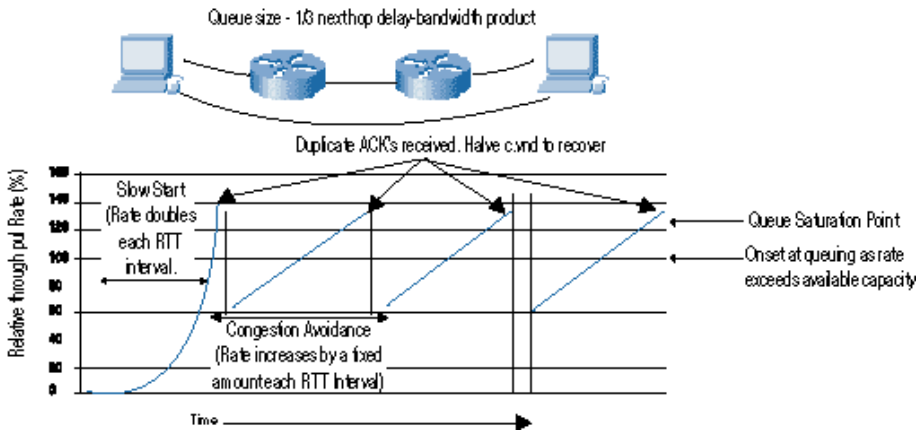
Compared to *Slow Start* , congestion avoidance is a more tentative probing of the network to discover the point of threshold of packet loss. Where *Slow Start* uses an exponential increase in the sending rate to find a first-level approximation of the loss threshold, congestion avoidance uses a linear growth function.

When the value of *cwnd* is greater than *ssthresh* , the sender increments the value of *cwnd* by the value *SMSS X SMSS/cwnd* , in response to each received nonduplicate ACK [7], ensuring that the congestion window opens by one segment within each RTT time interval.

The congestion window continues to open in this fashion until packet loss occurs. If the packet loss is isolated to a single packet within a packet sequence, the resultant duplicate ACKs will trigger the sender to halve the sending rate and continue a linear growth of the congestion window from this new point, as described above in fast recovery.

The behavior of *cwnd* in an idealized configuration is shown in Figure 8,

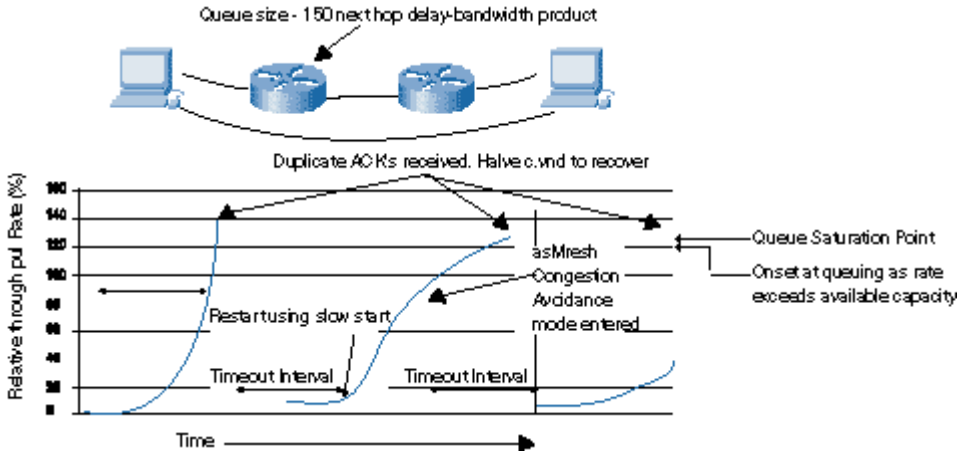
Figure 8: Simulation of Single TCP Transfer



along with the corresponding data-flow rates. The overall characteristics of the TCP algorithm are an initial relatively fast scan of the network capacity to establish the approximate bounds of maximal efficiency, followed by a cyclic mode of adaptive behavior that reacts quickly to congestion, and then slowly increases the sending rate across the area of maximal transfer efficiency.

Packet loss, as signaled by the triggering of the retransmission timer, causes the sender to recommence slow-start mode, following a timeout interval. The corresponding data-flow rates are indicated in Figure 9.

Figure 9: Simulation of TCP Transfer with Tail Drop Queue



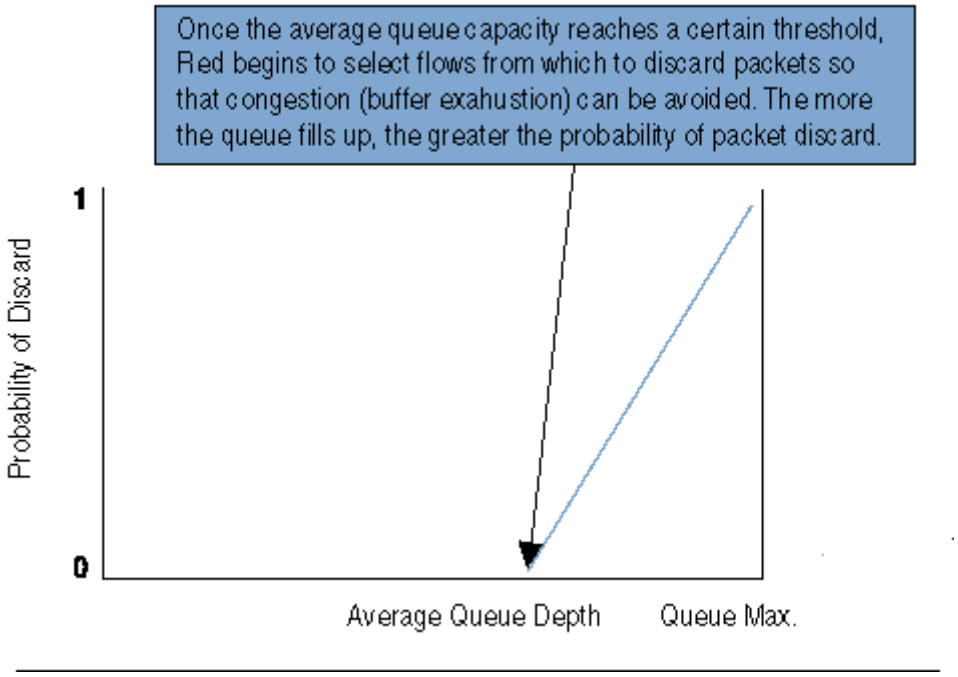
The inefficiency of this mode of performance is caused by the complete cessation of any form of flow signaling from the receiver to the sender. In the absence of any information, the sender can only assume that the network is heavily congested, and so must restart its probing of the network capacity with an initial congestion window of a single segment. This leads to the performance observation that any form of packet-drop management that tends to discard the trailing end of a sequence of data packets may cause significant TCP performance degradation, because such drop behavior forces the TCP session to continually time out and restart the flow from a single segment again.

Assisting TCP Performance Network-RED and ECN

Although TCP is an end-to-end protocol, it is possible for the network to assist TCP in optimizing performance. One approach is to alter the queue behaviour of the network through the use of *Random Early Detection* (RED). RED permits a network router to discard a packet even when there is additional space in the queue. Although this may sound inefficient, the interaction between this early packet-drop behaviour and TCP is very effective.

RED uses a the weighted average queue length as the probability factor for packet drop. As the average queue length increases, the probability of a packet being dropped, rather than being queued, increases. As the queue length decreases, so does the packet-drop probability. (See Figure 10). Small packet bursts can pass through a RED filter relatively intact, while larger packet bursts will experience increasingly higher packet-discard rates. Sustained load will further increase the packet-discard rates. This implies that the TCP sessions with the largest open windows will have a higher probability of experiencing packet drop, causing a back-off in the window size.

Figure 10: Red Behavior



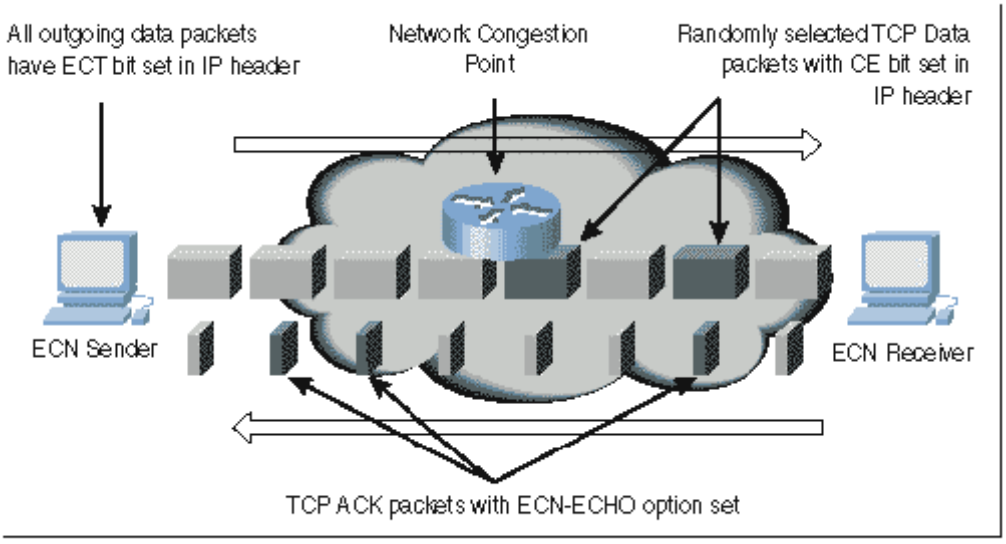
A major goal of RED is to avoid a situation in which all TCP flows experience congestion at the same time, all then back off and resume at the same rate, and tend to synchronize their behaviour [11,12]. With RED, the larger bursting flows experience a higher probability of packet drop, while flows with smaller burst rates can continue without undue impact. RED is also intended to reduce the incidence of complete loss of ACK signals, leading to timeout and session restart in slow-start mode. The intent is to signal the heaviest bursting TCP sessions the likelihood of pending queue saturation and tail drop before the onset of such a tail-drop congestion condition, allowing the TCP session to undertake a fast retransmit recovery under conditions of congestion avoidance. Another objective of RED is to allow the queue to operate efficiently, with the queue depth ranging across the entire queue size within a timescale of queue depth oscillation the same order as the average RTT of the traffic flows.

Behind RED is the observation that TCP sets very few assumptions about the networks over which it must operate, and that it cannot count on any consistent performance feedback signal being generated by the network. As a minimal approach, TCP uses packet loss as its performance signal, interpreting small-scale packet-loss events as peak load congestion events and extended packet loss events as a sign of more critical congestion load. RED attempts to increase the number of small-scale congestion signals, and in so doing avoid long-period sustained congestion conditions.

It is not necessary for RED to discard the randomly selected packet. The intent of RED is to signal the sender that there is the potential for queue exhaustion, and that the sender should adapt to this condition. An alternative mechanism is for the router experiencing the load to mark packets with an explicit *Congestion Experienced* (CE) bit flag, on the assumption that the sender will see and react to this flag setting in a manner comparable to its response to single packet drop [13] [14]. This mechanism, *Explicit Congestion Notification* (ECN), uses a 2-bit scheme, claiming bits 6 and 7 of the IP Version 4 Type-of-Service (ToS) field (or the two Currently Unused [CU] bits of the IP *Differentiated Services* field). Bit 6 is set by the sender to indicate that it is an ECN-capable transport system (the ECT bit). Bit 7 is the CE bit, and is set by a router when the average queue length exceeds configured threshold levels. The ECN algorithm is that an active router will perform RED, as described. After a packet has been

selected, the router may mark the CE bit of the packet if the ECT bit is set; otherwise, it will discard the selected packet. (See Figure 11).

Figure 11: Operation of Explicit Congestion Notification



The TCP interaction is slightly more involved. The initial TCP SYN handshake includes the addition of ECN-echo capability and *Congestion Window Reduced* (CWR) capability flags to allow each system to negotiate with its peer as to whether it will properly handle packets with the CE bit set during the data transfer. The sender sets the ECT bit in all packets sent. If the sender receives a TCP packet with the ECN-echo flag set in the TCP header, the sender will adjust its congestion window as if it had undergone fast recovery from a single lost packet.

The next sent packet will set the TCP CWR flag, to indicate to the receiver that it has reacted to the congestion. The additional caveat is that the sender will react in this way at most once every RTT interval. Further, TCP packets with the ECN-echo flag set will have no further effect on the sender within the same RTT interval. The receiver will set the ECN-echo flag in all packets when it receives a packet with the CE bit set. This will continue until it receives a packet with the CWR bit set, indicating that the sender has reacted to the congestion. The ECT flag is set only in packets that contain a data payload. TCP ACK packets that contain no data payload should be sent with the ECT bit clear.

The connection does not have to await the reception of three duplicate ACKs to detect the congestion condition. Instead, the receiver is notified of the incipient congestion condition through the explicit setting of a notification bit, which is in turn echoed back to the sender in the corresponding ACK. Simulations of ECN using a RED marking function indicate slightly superior throughput in comparison to configuring RED as a packet-discard function.

However, widespread deployment of ECN is not considered likely in the near future, at least in the context of Version 4 of IP. At this stage, there has been no explicit standardization of the field within the IPv4 header to carry this information, and the deployment base of IP is now so wide that any modifications to the semantics of fields in the IPv4 header would need to be very carefully considered to ensure that the changed field interpretation did not exercise some malformed behavior in older versions of the TCP stack or in older router software implementations.

ECN provides some level of performance improvement over a packet-drop RED scheme. With large bulk data transfers, the improvement is moderate, based on the difference between the packet retransmission and congestion-window adjustment of RED and the congestion-window adjustment of ECN. The most notable improvements indicated in ECN simulation experiments occur with short TCP transactions (commonly seen in Web transactions), where a RED packet drop of the initial data packet may cause a six-second retransmit delay. Comparatively, the ECN approach allows the transfer to proceed without this lengthy delay.

The major issue with ECN is the need to change the operation of both the routers and the TCP software stacks to accommodate the operation of ECN. While the ECN proposal is carefully constructed to allow an essentially uncoordinated introduction into the Internet without negative side effects, the effectiveness of ECN in improving overall network throughput will be apparent only after this approach has been widely adopted. As the Internet grows, its inertial mass generates a natural resistance to further technological change; therefore, it may be some years before ECN is widely adopted in both host software and Internet routing systems. RED, on the other hand, has had a more rapid introduction to the Internet, because it requires only a local modification to router behavior, and relies on existing TCP behavior to react to the packet drop.

Tuning TCP

How can the host optimize its TCP stack for optimum performance? Many recommendations can be considered. The following suggestions are a combination of those measures that have been well studied and are known to improve TCP performance, and those that appear to be highly productive areas of further research and investigation [1] .

- *Use a good TCP protocol stack* : Many of the performance pathologies that exist in the network today are not necessarily the byproduct of oversubscribed networks and consequent congestion. Many of these performance pathologies exist because of poor implementations of TCP flow-control algorithms; inadequate buffers within the receiver; poor (or no) use of path-MTU discovery; no support for fast-retransmit flow recovery, no use of window scaling and SACK, imprecise use of protocol-required timers, and very coarse-grained timers. It is unclear whether network ingress-imposed Quality-of-Service (QoS) structures will adequately compensate for such implementation deficiencies. The conclusion is that attempting to address the symptoms is not the same as curing the disease. A good protocol stack can produce even better results in the right environment.
- *Implement a TCP Selective Acknowledgment (SACK) mechanism* : SACK, combined with a selective repeat-transmission policy, can help overcome the limitation that traditional TCP experiences when a sender can learn only about a single lost packet per RTT.
- *Implement larger buffers with TCP window-scaling options* : The TCP flow algorithm attempts to work at a data rate that is the minimum of the delay-bandwidth product of the end-to-end network path and the available buffer space of the sender. Larger buffers at the sender and the receiver assist the sender in adapting more efficiently to a wider diversity of network paths by permitting a larger volume of traffic to be placed in flight across the end-to-end path.
- *Support TCP ECN negotiation* : ECN enables the host to be explicitly informed of conditions relating to the onset of congestion without having to infer such a condition from the reserve stream of ACK packets from the receiver. The host can react to such a condition promptly and effectively with a data flow-control response without having to invoke packet retransmission.
- *Use a higher initial TCP slow-start rate than the current 1 MSS (Maximum Segment Size) per RTT* . A size that seems feasible is an initial burst of 2 MSS segments. The assumption is that there will be adequate queuing capability to manage this initial packet burst; the provision to back off the send window to 1 MSS segment should remain intact to allow stable operation if the initial choice was too large for the path. A robust initial choice is two segments, although simulations have indicated that four initial segments is also highly effective in many situations.

- *Use a host platform that has sufficient processor and memory capacity to drive the network* . The highest-quality service network and optimally provisioned access circuits cannot compensate for a host system that does not have sufficient capacity to drive the service load. This is a condition that can be observed in large or very popular public Web servers, where the peak application load on the server drives the platform into a state of memory and processor exhaustion, even though the network itself has adequate resources to manage the traffic load.

All these actions have one thing in common: They can be deployed incrementally at the edge of the network and can be deployed individually. This allows end systems to obtain superior performance even in the absence of the network provider tuning the network's service response with various internal QoS mechanisms.

Conclusion

TCP is not a predictive protocol. It is an adaptive protocol that attempts to operate the network at the point of greatest efficiency. Tuning TCP is not a case of making TCP pass more packets into the network. Tuning TCP involves recognizing how TCP senses current network load conditions, working through the inevitable compromise between making TCP highly sensitive to transient network conditions, and making TCP resilient to what can be regarded as noise signals.

If the performance of end-to-end TCP is the perceived problem, the most effective answer is not necessarily to add QoS service differentiation into the network. Often, the greatest performance improvement can be made by upgrading the way that hosts and the network interact through the appropriate configuration of the host TCP stacks.

In the next article on this topic, we will examine how TCP is facing new challenges with increasing use of wireless, short-lived connections, and bandwidth-limited mobile devices, as well as the continuing effort for improved TCP performance. We'll look at a number of proposals to change the standard actions of TCP to meet these various requirements and how they would interact with the existing TCP protocol.

References

[1] Huston, G., *Internet Performance Survival Guide* : QoS Strategies for Multiservice Networks, ISBN 0471-378089, John Wiley & Sons, January 2000.

[2] Postel, J., "Transmission Control Protocol," RFC 793, September 1981. [3] Jacobson, V., Braden, R., and Borman, D., "TCP Extensions for High Performance," RFC 1323, May 1992.

[4] Mathis, M., Madavi, J., Floyd, S., and Romanow, A., "TCP Selective Acknowledgement Options," RFC 2018, October 1996.

[5] Nagle, J., "Congestion Control in IP/TCP Internetworks," RFC 896, January 1984.

[6] Allman, M., Floyd, S., and Partridge, C., "Increasing TCP's Initial Window," RFC 2414, September 1998.

[7] Allman, M., Paxson, V., and Stevens, W., "TCP Congestion Control," RFC 2581, April 1999.

[8] Stevens, W. R., TCP/IP Illustrated, Volume 1, Addison-Wesley, 1994.

[9] Jacobson V., "Congestion Avoidance and Control," *ACM Computer Communication Review* , Vol. 18, No. 4, August 1988.

[10] Jacobson, V., "Berkeley TCP Evolution from 4.3-Tahoe to 4.3, Reno," Proceedings of the 18th Internet Engineering Task Force, University of British Colombia, Vancouver, BC, September 1990.

[11] Floyd, S., and Jacobson, V., "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Transactions on Networking* , Vol. 1, No. 4, August 1993.

[12] Braden, R. et al., "Recommendations on Queue Management and Congestion Avoidance in the Internet," RFC 2309, April 1998.

[13] Floyd, S., "TCP and Explicit Congestion Notification," *ACM Computer Communication Review* , Vol. 24, No. 5, October 1994.

[14] Ramakrishnan, K., and Floyd, S., "A Proposal to Add Explicit Congestion Notification (ECN) to IP," RFC 2481, January 1999.

GEOFF HUSTON holds a B.Sc. and a M.Sc. from the Australian National University. He has been closely involved with the development of the Internet for the past decade, particularly within Australia, where he was responsible for the initial build of the Internet within the Australian academic and research sector. Huston is currently the Chief Technologist in the Internet area for Telstra. He is also an active member of the IETF, and is the chair of the Internet Society Board of Trustees. He is author of The *ISP Survival Guide* , ISBN 0-471-31499-4, *Internet Performance Survival Guide: QoS Strategies for Multiservice Networks* , ISBN 0471-378089, and coauthor of *Quality of Service: Delivering QoS on the Internet and in Corporate Networks* , ISBN 0-471-24358-2, a collaboration with Paul Ferguson. All three books are published by John Wiley & Sons. E-mail: gih@telstra.net