Evolving TCP

August 2004

The Transmission Control Protocol, or TCP is certainly the workhorse of the IP protocol suite. IP alone is a basic datagram service: packets may be lost, duplicated, reordered or corrupted during their transit across a network path. While this allows IP to be placed across a huge variety of transmission media, it does not make for a very useful communications service. TCP is the transport adaptation layer, taking IP datagrams and applying an end-to-end protocol interaction that creates as an outcome a reliable end-to-end data transfer function that includes a number of control functions that are intended to make efficient use of the underlying network path through a host-based congestion control function.

Today’s Internet spans a very broad base of uses, and ensuring that TCP provides a highly robust, efficient, and reliable service platform for such a diversity of use is a continuing task. The Web generates a large component of short duration reliable transfers. These short sessions are often referred to as “TCP mice” because of the short duration and large number of such TCP sessions. Complementing these short sessions is the increasing size of large transfers as File Transfer Protocol data sets commonly associated with multimedia files. In addition, there is an increasing diversity of media used within the Internet, both in terms of higher-speed systems and in the use of wireless systems for Internet access.

In this article we will look at how TCP is being used and adapted to match this changing environment.

A Review of TCP Operation

Within any packet-switched network, when demand exceeds available capacity at a switching point, the packet switch will use a queue to hold the excess packets. When this queue fills, the packet switch must drop packets. Any reliable data protocol that operates across such a network must recognize this possibility and take corrective action. TCP is no exception to this constraint. TCP uses data sequence numbering to identify packets, and explicit acknowledgements (ACKs) to allow the sender and receiver to be aware of reliable packet transfer. This form of reliable protocol design is termed “end-to-end” control, as distinct to "hop-by-hop" control, as interior switches do not attempt to correct packet drops at the point of the drop. Instead, this function is performed through the TCP protocol exchange between sender and receiver. TCP uses cumulative ACKs rather than per-packet ACKs, where an ACK referencing a particular point within the data stream implicitly acknowledges all data with a sequence value less than the ACKed sequence value.

TCP uses these ACKs to clock the data flow. ACKs arriving back at the sender arrive at intervals approximately equal to the intervals at which the data packets arrived at the sender. If TCP uses these ACKs to trigger sending further data packets into the network, then the packets will be entered into the network at the same rate as they are arriving at their destination. This mode of operation is termed “ACK clocking.” In terms of supporting network stability this is a very astute design decision. In a steady state this mechanism will ensure that TCP injects one packet into the network at the same time as one packet is removed at the other end.

TCP recovers from packet loss using two mechanisms. The most basic operation is the use of packet timeouts by the sender. If an ACK for a packet fails to arrive within the timeout value, the sender will retransmit the oldest unacknowledged packet. In such a case, TCP assumes that the loss was caused by a network congestion condition, and the sender will enter “Slow Start” mode. Slow Start is a mode where the sender cuts back the amount of unacknowledged data in flight across the network path to a zero base, and recommences by sending a single packet and waiting for the corresponding ACK, and then increasing its data rate from this initial state once more. This condition causes significant delays within the data transfer, because the sender will be idle during the timeout interval and upon restarting will recommence with a single packet exchange, gradually recovering the data rate that was active prior to the packet loss. Many networks exhibit transient congestion conditions, where a data stream may experience loss of a single packet within a packet train. To address this, TCP introduced the mechanism of “fast recovery.” This mechanism is triggered by a sequence of three duplicate ACKS received by the data sender. These duplicate ACKs are generated by the packets that trail the lost packet, where the sender ACKs each of these packets with the ACK sequence value of the lost packet. In this mode the sender immediately retransmits the lost packet and then halves its sending rate, continuing to send additional data as permitted by the current TCP sending window. In this mode of operation, “congestion-avoidance” TCP increases its sending window at a linear rate of one segment per Round-Trip Time (RTT). This mode of operation is referred to as Additive Increase, Multiplicative Decrease (AIMD), where the protocol reacts sharply to signs of network congestion, and gradually increases its sending rate in order to equilibrate with concurrent TCP sessions.

TCP Design Assumptions

It is difficult to design any transport protocol without making some number of assumptions about the environment in which the protocol is to be used, and TCP certainly has some inherent assumptions hidden within its design. The most important set of assumptions that lie behind the design of TCP are as follows:

- **A network of wires, not wireless**: As we continually learn, wireless is different. Wireless systems typically have higher bit error rates (BERs) than wire-based carriage systems. Mobile wireless systems also include factors of signal fade, base-station handover, and variable levels of load. TCP was designed with wire-based carriage in mind, and the design of the protocol makes numerous assumptions that are typical of such of an environment. TCP makes the assumption that packet loss is the result of network congestion, rather than bit-level corruption. TCP also assumes some level of stability in the RTT, because TCP uses a method of damping down the changes in the RTT estimate.
- **A best-path route-selection protocol**: TCP assumes that there is a single best metric path to any destination because TCP assumes that packet reordering occurs on a relatively minor scale, if at all. This implies that all packets in a connection must follow the same path within the network or, if there is any form of load balancing, the order of packets within each flow is preserved by some network-level mechanism.
- **A network with fixed bandwidth circuits, not rapidly varying bandwidth**: TCP assumes that available bandwidth is constant, and will not vary over short time intervals. TCP uses an end-to-end control loop to control the sending rate, and it takes many RTT intervals to adjust to varying network conditions. Rapidly changing bandwidth forces TCP to make very conservative assumptions about available network capacity.
- **A switched network with first-in, first-out (FIFO) buffers**: TCP also makes some assumptions about the architecture of the switching elements within the network. In particular, TCP assumes that the switching elements use simple FIFO queues to resolve contention within the switches. TCP makes some assumption about the size of the buffer as well as its queuing behavior, and TCP works most efficiently when the buffer associated with a network interface is of the same order of size as the delay bandwidth product of the associated link.
- **Non-trivial sessions**: TCP also makes some assumptions about the nature of the application. In particular, it assumes that the TCP session will last for some number of round-trip times, so that the overhead of the initial protocol handshake is not detrimental to the efficiency of the application. TCP also takes numerous RTT intervals to establish the characteristics of the connection in terms of the true RTT interval of the connection as well as the available capacity. The introduction of short-duration sessions, such as found in transaction applications and short Web transfers, is a new factor that impacts the efficiency of TCP.
- **Large payloads and adequate bandwidth**: TCP assumes that the overhead of a minimum of 40 bytes of protocol per TCP packet (20 bytes of IP header and 20 bytes of TCP header) is an acceptable overhead when compared to the available bandwidth and the average payload size. When applied to low-bandwidth links, this is no longer the case, and the protocol overheads may make the resultant communications system too inefficient to be useful.
- **Interaction with other TCP sessions**: TCP assumes that other TCP sessions will also be active within the network, and that each TCP session should operate cooperatively to share available bandwidth in order to

maximize network efficiency. TCP may not interact well with other forms of flow-control protocols, and this could result in unpredictable outcomes in terms of sharing of the network resource between the active flows as well as poor overall network efficiency. If these assumptions are challenged, the associated cost is that of TCP efficiency. If the objective is to extend TCP to environments where these assumptions are no longer valid, while preserving the integrity of the TCP transfer and maintaining a high level of efficiency, then the TCP operation itself may have to be altered.

There are two basic ways of altering TCP operation: by altering the actions of the end host by making changes to the TCP protocol, or by altering the characteristics of the network, making them more “friendly” to TCP. We will look at the potential for both responses in examining various scenarios for adapting TCP to suit these changing environments.

Some caution should be noted about making changes to the TCP protocol. The major constraint is that any changes that are contemplated to TCP should be backward compatible with existing TCP behavior. This constraint requires a modified TCP protocol to attempt to negotiate the use of a specific protocol extension, and the knowledge that a basic common mode of protocol operation may be required if the negotiation fails. The second constraint is that TCP assumes that it is interacting with other TCP sessions within the network, and the outcome of fair sharing of the network between concurrent sessions depends on some commonality of the protocol used by these sessions. Major changes to the protocol behavior can lead to unpredictable outcomes in terms of sharing of the network resource between “unmodified” and “modified” TCP sessions, and unpredictable outcomes in terms of efficiency of the use of the network. For this reason there is some understandable reluctance to undertake modifications of TCP that radically alter TCP startup behavior or behavior in the face of network congestion.

Short-Duration Sessions — TCP for Transactions

For network applications that generate small transactions, the application designer is faced with a dilemma. The application may be able to use the User Datagram Protocol (UDP), in which case the sender must send the query and await the response as a single data exchange. This operation is highly efficient, as the total elapsed time for the client is a single RTT. However, this speed is gained at the cost of reliability. A missing response is ambiguous, in that it is impossible for the initiator to tell whether the query was lost or the response was lost. If multiple queries are generated, it is not necessarily true that they will arrive at the remote server in the same order as they were generated, and it may not be possible to correctly match query to response.

Alternatively, the application can use TCP, which will ensure reliability of the transaction. However, TCP uses a three-way handshake to complete the opening of the connection, and uses acknowledged FIN signals for each side to close its end of the connection after it has completed sending data. Under the control of TCP, the sender will retransmit the query until it receives an acknowledgment that the query has arrived at the remote server. Similarly, the remote host will retransmit the response until the server receives an indication that the response has been successfully delivered. The cost of this reliability is application efficiency, as the minimum time to conduct the TCP transaction for the client is two RTT intervals.

TCP for Transactions (commonly referred to as T/TCP) [RFC1644] attempts to improve the performance of small transactions while preserving the reliability of TCP. T/TCP places the query data and the closing FIN in the initial SYN packet. This can interpreted as attempting to open a session, pass data, and close the sender’s side of the session within a single packet. If the server accepts this format, the server responds with a single packet, which contains its SYN response, an ACK of the query data, the server’s data in response, and the closing FIN. All that is required to complete the transaction is for the query system to ACK the server’s data and FIN (Figure 1).

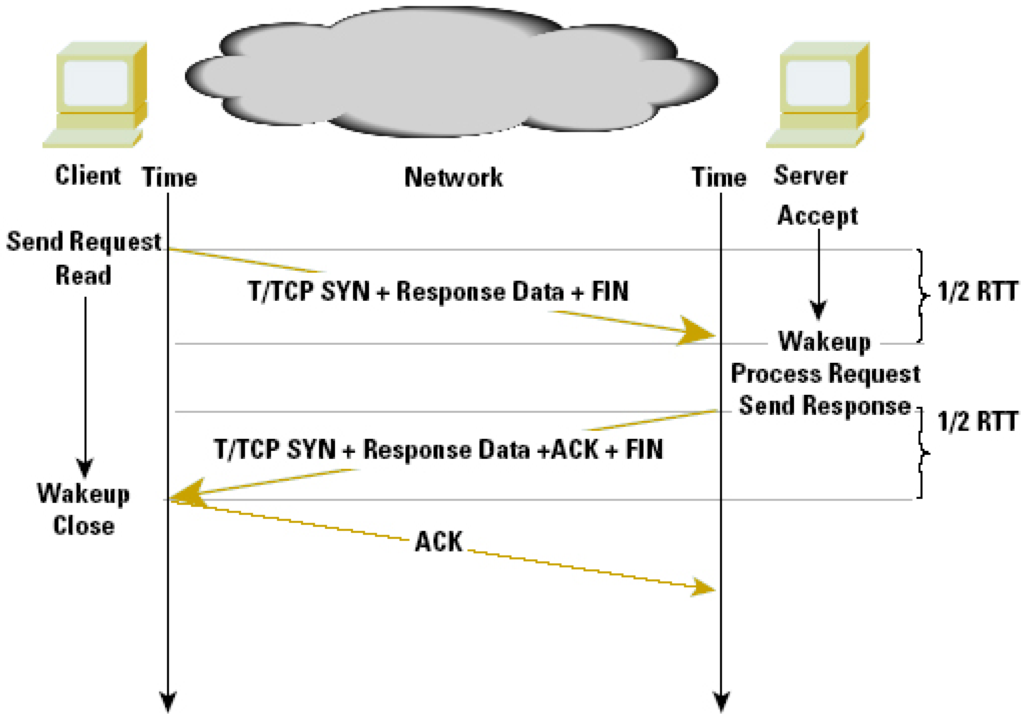

Figure 1 – T/TCP

If the server does not accept this format, the client can back off to a conventional TCP handshake followed by a data exchange.

For the client, the time to undertake this T/TCP transaction is one RTT interval, a period equal to the UDP-supported transaction, while still allowing for the two systems to use TCP to negotiate a reliable exchange of data as a backup.

T/TCP requires changes to the protocol stack of both the sender and the receiver in order to operate correctly. The design of the protocol explicitly allows the session initiator to back off to use TCP if the receiver cannot correctly respond to the initial T/TCP packet. T/TCP is not in common use in the Internet today, because while it improves the efficiency of simple transactions, the limited handshake makes it more vulnerable from a security perspective. , and various concerns over this vulnerability appear to have been a prohibitive factor in its adoption. This is illustrative of the nature of the trade-offs that occur within protocol design, where optimizing one characteristic of a protocol may be at the expense of other aspects of the protocol.

Long Delay — TCP for Satellite Paths

Satellite-based services pose a set of unique issues to the network designer. Most notably, these issues include delay, bit errors, and bandwidth.

When using a satellite path, there is an inherent delay in the delivery of a packet due to signal propagation times related to the altitude of communications satellites. Geo-stationary orbit spacecraft are located at an altitude of some 36,000 km, and the propagation time for a signal to pass from an earth station directly below the satellite to the satellite and back is 239.6 ms. If the earth station is located at the edge of the satellite view area, this propagation time extends to 279.0 ms. In terms of a round trip that uses the satellite path in both directions, the RTT of a satellite hop is between 480 and 560 ms.

The strength of a radio signal falls in proportion to the square of the distance traveled. For a satellite link, the signal propagation distance is large, so the signal becomes weak before reaching its destination, resulting in a poor signal-to-noise ratio. Typical BERs for a satellite link today are on the order of 1 error per 10 million bits. Forward error correction (FEC) coding can be added to satellite services to reduce this error rate, at the cost of some reduction in available bandwidth and an increase in latency due to the coding delay. There is also a limited amount of bandwidth available to satellite systems. Typical carrier frequencies for commercial satellite services are 6/4 GHz (C-band) and 14/12 GHz (Ku band). Satellite transponder bandwidth is typically 36 MHz [RFC2448] .

When used in a data carriage role for IP traffic, satellite channels pose several challenges for TCP.

The delay-bandwidth product of a transmission path defines the amount of data TCP should have within the transmission path at any one time, in order to fully utilize the available channel capacity. The delay used in this equation is the RTT and the bandwidth is the capacity of the bottleneck link in the network path. Because the delay in satellite environments is large, a TCP flow may need to keep a large amount of data within the transmission path. For example, a typical path that includes a satellite hop may have a RTT of some 700 ms. If the bottleneck bandwidth is 2 Mbps, then a sender will need to buffer 180 kB of data to fully utilize the available bandwidth with a single traffic flow. For this to be effective, the sender and receiver will need to agree on the use of TCP Window Scaling to extend the available window size beyond the protocol default limit of 64 kB. A sender using an 8 kB buffer would be able to achieve a maximum transfer rate of 91 kbps, irrespective of the available bandwidth on the satellite path.

Even with advanced FEC techniques, satellite channels exhibit a higher BER than typical terrestrial networks. TCP interprets packet drop as a signal of network congestion, and reduces its window size in an attempt to alleviate the situation. In the absence of certain knowledge about whether a packet was dropped because of congestion or corruption, TCP must assume the drop was caused by congestion in order to avoid congestion collapse [CTL] [E2ECTL] . Therefore, packets dropped because of corruption cause TCP to reduce the size of its sending window, even though these packet drops do not signal congestion in the network. To mitigate this, some care must be taken with the satellite hop Maximum Transmission Unit (MTU) size, to reduce the probability of packet corruption. This is an area of compromise, in that the consequence is the potential for a high level of IP packet fragmentation on the satellite feeder router. In addition, the sender needs to use the TCP fast retransmit and fast recovery algorithms [RFC2581] in order to recover from the packet loss in a rapid, but stable fashion. In addition, the sender needs to use larger sending windows to operate the path more efficiently, with a consequent risk of multiple packet drops per RTT window. For this reason the use of Selective Acknowledgements (SACKs) is necessary in order to recover from multiple packet drops in a single RTT interval.

The long delay causes TCP to react slowly to the prevailing conditions within the network. The slow start of TCP commences with a single packet exchange, and it takes some number of RTT intervals for the sender’s rate to reach the same order of size as the delay bandwidth product of the long delay path. For short-duration TCP transactions, such as much of the current Web traffic, this is a potential source of inefficiency. For example, if a transaction requires the transfer of ten packets, the slow-start algorithm will send a single packet in the first RTT interval, two in the second interval, four in the third, and the remaining three packets in the fourth RTT interval. Irrespective of the available bandwidth of the path, the transaction will take a minimum of four RTT intervals. This theoretical model is further exacerbated by delayed ACKs [RFC 1122], where a receiver will not immediately ACK a packet, but will await the expiration of the 500ms ACK timer, or a second full-sized packet. During slow start, where a sender sends an initial packet, and then awaits an ACK, the receiver will delay the ACK until the expiration of the delayed ACK timer, adding up to 500ms additional delay in the first data exchange. The second part of the delayed ACK algorithm is that it will only ACK every second full-sized data packet, slowing down the window inflation rate of slow start. Also, if congestion occurs on the forward data path, the TCP sender will not be aware of the condition until it receives duplicate ACKs from the receiver. A congestion condition may take many RTT intervals to clear, and in the case of a satellite path, transient congestions may take tens of seconds to be resolved.

The TCP mechanisms that assist in mitigating some of the more serious effects of satellite systems include Path MTU Discovery [RFC1191] , Fast Retransmit and Fast Recovery, window scaling options, in order to extend the sender’s buffer beyond 65,535 bytes [RFC1323] , and the companion mechanisms of Protection Against Wrapped Sequence Space (PAWS) and Round-Trip Time Measurements (RTTM) and SACKs [RFC2018] . A summary of TCP options is shown in Figure 2.

Mechanism	Use	Location
Path-MTU Discovery	Recommended	Sender
FEC	Recommended	Link
TCP		
Slow Start	Required	Sender
Congestion Avoidance	Required	Recommended Sender
Fast Retransmit	Recommended	Recommended Sender
Fast Recovery	Recommended	Recommended Sender
Window Scaling	Recommended	Sender and Receiver
PAWS	Recommended	Sender and Receiver
RTTM	Recommended	Sender and Receiver
SACK	Recommended	Sender and Receiver

Figure 2: TCP Options for Satellite Paths (after [RFC 2488])

Further refinements to the TCP stack have been considered in relation to satellite performance [RFC2760] .

The options considered include the use of T/TCP as a means of reducing the overhead of the initial TCP three-way handshake. This is effective for short transactions where the data to be transferred can be held in a single packet, or in a small number of packets.

The use of delayed acknowledgements also is an issue for long-delay network paths, particularly if the sender is using slow start with an initial window of a single segment. In this case, the receiver will not immediately acknowledge the initial packet, but will wait up to one-half second for the delayed ACK timer to trigger. Altering the initial window size to two segments allows the receiver to trigger an ACK on reception of the second packet, bypassing the delayed ACK timer. However, even this change to TCP does not completely address the performance issue relating to delayed ACKs on long delay paths for TCP slow start. The delayed ACK algorithm triggers an ACK on every second full-sized packet. Because the sender's congestion window is opened on receipt of ACKs, this causes the slow-start window to open more slowly than if the receiver generated an ACK every packet. One variant of TCP congestion control allows the TCP sender to count the number of bytes acknowledged in an ACK message to control the expansion of the congestion window, making the algorithm less sensitive to delayed ACKs [RFC2581] . Although this approach has some merit for long delay paths, this is a case where the correction is potentially as bad as the original problem. The byte counting mode of congestion control allows a sender to sharply increase its sending rate, causing potential instabilities within the network and impacting concurrent TCP sessions.

One approach to address this is to place a limit on the size of the window expansion, where each increment of the congestion window is limited to the minimum of one or two segment sizes and the size of the data spanned by the ACK. If the limit is set to a single segment size, the window expansion will be in general slightly more conservative to the current TCP ACK-based expansion mechanism. If this upper limit is set to two segments, the congestion window expansion will account for the delayed ACKs, expand at a rate equal to one segment for every successfully transmitted segment during slow start, and expand the window by one segment size each RTT during congestion avoidance. Because a TCP receiver will ACK a large span of data following recovery, this byte counting is bounded to a single segment per ACK in the slow-start phase following a transmission timeout. Another approach that has been explored is for the receiver to disable delayed ACKs until the sender has completed the slow-start phase. Although such an approach shows promising results under simulated conditions, the practical difficulty is that it is difficult for the receiver to remotely determine the current TCP sending state, and the receiver cannot reliably tell if the sender is in slow start, congestion avoidance, or in some form of recovery mode. Explicit signaling of the sender's state as a TCP flag is an option, but the one-half RTT delay in the signaling from the sender to the receiver may prove to be an issue here. This area of congestion control for TCP remains a topic of study.

All of these approaches can mitigate only the worst of the effects of the long delay paths. TCP, as an adaptive reliable protocol that uses end-to-end flow control, can undertake only incremental adjustments in its flow rates in intervals of round-trip times. When the round-trip times extend, then TCP is slower to speed up from an initial start, slower to recover from packet loss, and slower to react to network congestion.

Tuning TCP—ACK Manipulation

The previous article on TCP discussed numerous network responses to congestion using Random Early Detection (RED) for active queue control and Explicit Congestion Notification (ECN) as an alternative to RED packet drop. It is feasible for a network control point to impose a finer level of control on a TCP flow by using an approach of direct manipulation of the TCP packets.

The approaches described above to mitigate some of the side effects of satellite paths all share in the side effect of having some latency associated with the congestion response. The sender must await the reception of trailing packets by the receiver, and then await the reception of the matching ACK packets from the data receiver back to the sender to learn of the fate of the original data packet. This may take up to one RTT interval to complete. An alternative approach to congestion management responses is to manipulate the ACK packets to modify the sender's behavior.

The prerequisite to perform this manipulation is that the traffic path be symmetric, so that the congestion point can identify ACK packets traveling in the opposite direction. If this is the case, a couple of control alternatives can mitigate the onset of congestion:

- **ACK Pacing**: Each burst of data packets will generate a corresponding burst of ACK packets. The spacing of these ACK packets determines the burst rate of the next sending packet sequence. For long-delay systems, the size of such bursts becomes a limiting factor. TCP slow start generates packet bursts at twice the bottleneck data rate, so that the bottleneck feeder router may have to absorb one-half of every packet burst within its internal queues. If these queues are not dimensioned to the delay bandwidth product of the next hop, these queues become the limiting factor, rather than the path bandwidth itself. If you can slow down the TCP burst rate, the pressure on the feeder queue is alleviated. One approach to slow down the burst rate is to impose a delay on successive ACKs at a network control point (Figure 3). This measure will reduce the burst rate, but not impact the overall TCP throughput. ACK pacing is most effective on long delay paths, and it is intended to spread out the burst load, reducing the pressure on the bottleneck queue and increasing the actual data throughput.

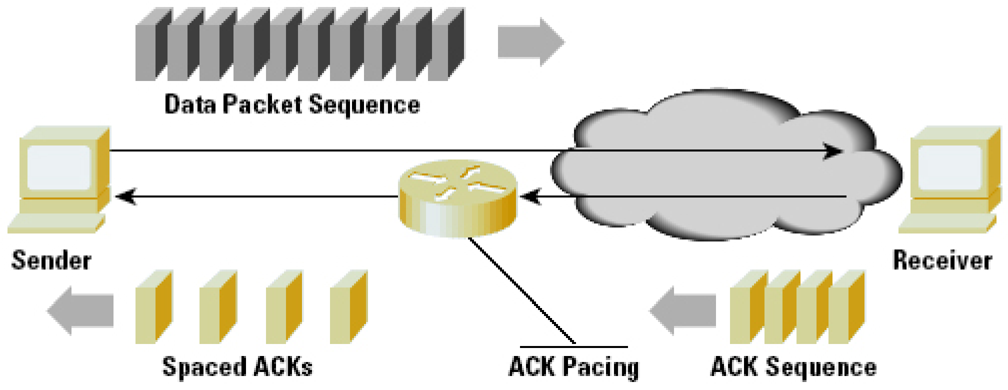

Figure 3: ACK Pacing

- **Window Manipulation**: Each ACK packet carries a receiver window size. This advertised window determines the maximum burst size available to the sender. Manipulating this window size downward allows a control point to control the maximal TCP sending rate. This manipulation can be done as part of a traffic-shaping control point, enforcing bandwidth limitations on a flow or set of flows. Both of these mechanisms make some sweeping assumptions about the network control point that must be carefully understood. The major assumption is that these mechanisms assume symmetry of data flows at the network control point, where the data and the associated ACKs flow through this control point (but in opposite directions, of course).

Both mechanisms also assume that the control point can cache per-flow state information, so that the current flow RTT and the current transfer rate and receiver window size are available to the service controller. Sender Receiver Data Packet Sequence Spaced ACKs ACK Pacing ACK Sequence

ACK pacing also implicitly assumes that a single ACK timing response is active at any time along a network path. A sequence of ACK delay actions may cause the sender’s timers to trigger, and the sender to close down the transfer and reenter slow-start mode. These environmental conditions are more common at the edge of the network, and such mechanisms are often part of a traffic control system for Web-hosting platforms or similar network service delivery platforms. As a network control tool, ACK manipulation makes too many assumptions, and the per-flow congestion state information represents a significant overhead for large network systems. In general, such manipulations are more appropriate as an edge traffic filter, rather than as an effective congestion management response. For this reason, the more indirect approach of selective data packet discard is more effective as a congestion management measure.

Assisting Short-Duration TCP Sessions — Limited Transmit

One of the challenges to the original set of TCP assumptions is that of short-duration TCP sessions. The Web has introduced a large number of short-duration sessions, and the issue with these sessions is that they use small initial windows. If congestion loss occurs within this early period of TCP slow start, there are not enough packets in the network to generate the three duplicate ACKs required to initiate fast retransmit and fast recovery. Instead the TCP sender must await the expiry of the retransmission timeout (RTO), a timer that uses a minimum value of one second. For short-duration TCP sessions that may last six or seven RTT intervals of a small number of milliseconds, the incremental penalty of single packet loss is then extremely severe. A study of this problem indicates that approximately 56 percent of retransmissions are sent following an RTO timeout [RFC 3042].

One potential mitigation to this is a mechanism termed “Limited Transmit.” With this mechanism, a duplicate ACK may trigger an immediate transmission of a segment of new data. Two conditions are applied to this; the receiver’s advertised window allows the transmission of this segment, and the amount of outstanding data would remain less than the congestion window plus the duplicate ACK threshold used to trigger Fast Retransmit. This second condition implies that the sender can send only two segments beyond the congestion window, and will do so only in response to the receiver lifting a segment off the network. The basic principle of this strategy is to continue the signaling between the sender and receiver in the face of packet loss, increasing the probability that the sender will recover from packet loss using duplicate ACKs and fast recovery, and reducing the probability of the one-second (or longer) RTO timeout as being the recovery trigger. The limited transmit also reduces the potential for the recovery actions to burst into the network at a level that may cause further packet loss.

Low Bandwidth and Higher Error Rates — TCP for Wireless Systems

One of the more challenging environments for a the Internet Protocol, and TCP in particular, is that of mobile wireless.

One approach to supporting the wireless environment is that of the so- called “walled garden.” Here the protocols in use within the wireless environment are specifically adapted to the wireless world. The transport protocols can account for the low bandwidth, the longer latency, the BERs, and the variability within all three of these metrics. In this model, Internet applications interact with an application gateway to reach the wireless world, and the application gateway uses a wireless transport protocol and potentially a modified version of the application data to interact with the mobile wireless device. The most common approach is interaction of Internet-based services with a wireless realm using some form of proxy server at the boundary of the wireless network and the Internet.

An alternative approach lies in allowing mobile devices to access a complete range of Internet-based services as the functional objective. In this approach, the intent is to allow the mobile wireless device to function as any other Internet-connected device, and there is a consequent requirement for some form of end-to-end direct IP continuity, and an associated requirement for end-to-end TCP functionality, where the TCP path straddles both wired and wireless segments. Ensuring the efficient operation of TCP in this environment is an integral part of the development of such an environment. Given that TCP must now work within a broader environment, it is no longer a case of adjusting TCP to match the requirements of the wireless environment, but one of attempting to provide seamless interworking between the wired and wireless worlds.

The wireless environment challenges many of the basic assumptions of TCP noted above. Wireless has significant levels of bit error rates, often with bursting of very high error rates. Wireless links that use forward error correcting codes have higher latency. If the link level protocol includes automatic retransmission of corrupted data, this latency will have high variability. Wireless links may also use adaptive coding techniques that adjust to the prevailing signal to noise ratio of the link, in which case the link will have varying bandwidth. If the wireless device is a hand-held mobile device, it may also be memory constrained. And finally, such an environment is typically used to support short duration TCP sessions.

The major factor for mobile wireless is the BER, where frame loss of up to 1 percent is not uncommon, and errors occur in bursts, rather than as evenly spaced bit errors in the packet stream. In the case of TCP, such error conditions force the TCP sender to initially attempt fast retransmit of the missing segments, and when this does not correct the condition, the sender will have an ACK timeout occur, causing the sender to collapse its sending window and recommence from the point of packet loss in slow-start mode. The heart of this problem is that assumption on the part of TCP that packet loss is a symptom of network congestion rather than packet corruption. It is possible to use a model of TCP AIMD performance to determine the effects of this loss rate on TCP performance. If, for example the link has a 1-percent average packet loss rate, a Maximum Segment Size (MSS) size of 1000 bytes, and a 120ms RTT, then the AIMD models predict a best-case performance of 666Kbps throughput, and a more realistic target of 402Kbps throughput [PILC] . (See the appendix for details of these models.) TCP is very sensitive to packet loss levels, and sustainable performance rapidly drops when packet drop levels exceed 1 percent.

Link-level solutions to the high BER are available to designers, and FEC codes and automatic retransmission systems (ARQ) can be used on the wireless link. FEC introduces a relatively constant coding delay and a bandwidth overhead into the path, but cannot correct all forms of bit error corruption. ARQ uses a “stop and resend” control mechanism similar to TCP itself. The consequent behavior is one of individual packets experiencing extended latency as the ARQ mechanisms retransmit link-level fragments to correct the data corruption, because the packet flow may halt for an entire link RTT interval for the link-level error to be signaled and the corrupted level 2 data to be retransmitted. The issue here is that TCP may integrate these extended latencies into its RTT estimate, making TCP assume a far higher latency on the path than is the case, or, more likely, it may trigger a retransmission at the same time as the level 2 ARQ is already retransmitting the same data. An alternative Layer 2 approach to bit-level corruption is to deliver those level 2 frames that were successfully transmitted, while resending any frames that were corrupted in transmission.

The problem for TCP here is that the level 2 drivers are adding packet reordering to the extended latency, and from TCP perspective the delivery of the out-of-order packets will generate duplicate ACKs that may trigger a simultaneous TCP fast retransmit.

Perversely, some approaches have advocated TCP delaying its duplicate ACK response in such situations [RFC2760] . To quote from RFC 2488, “The interaction between link-level retransmission and transport-level retransmission is not well understood.” [RFC2448]

If ARQ is not the best possible answer to addressing packet loss in mobile wireless systems, then what can be done at the TCP level to address this? TCP can take numerous basic steps to alleviate the worst aspects of packet corruption on TCP performance. These include the use of Fast Retransmit and Fast Recovery to allow a single packet loss to be repaired moderately quickly. This mechanism triggers only after three duplicate ACKs, so the associated action is to ensure that the TCP sender and receiver can advertise buffers of greater than four times the MSS. SACKs allow a sender to repair multiple segment losses per window within a single RTT, and where large windows are operated over long delay paths, SACK is undoubtedly useful.

However, useful as these mechanisms may be, they are probably inadequate to allow TCP to function efficiently over all forms of wireless systems. Particularly in the case of mobile wireless systems, packet corruption is sufficiently common that, for TCP to work efficiently, some form of explicit addressing of network packet corruption appears to be necessary.

One approach is to decouple TCP congestion control mechanisms from data recovery actions. The intent is to allow new data to be sent during recovery to sustain TCP ACK clocking. This approach is termed Forward Acknowledgements with Rate Halving (FACK) [RFC2760] , where one packet is sent for every two ACKs received while TCP is recovering from lost packets. This algorithm effectively reduces the sending rate by one-half within one RTT interval, but does not freeze the sender to wait the draining on one-half of the congestion window's amount of data from the network before proceeding to sending further data, nor does it permit the sender to burst retransmissions into the network. This is particularly effective for long-delay networks, where the fast recovery algorithm causes the sender to cease sending for up to one RTT interval, thereby losing the accuracy of the implicit ACK clock for the session. FACK allows the sender to continue to send packets into the network during this period, in an effort to allow the sender to maintain an accurate view of the ACK clock. FACK also provides an ability to set the number of SACK blocks that specify a missing segment before resending the segment, allowing the sender greater levels of control over sensitivity to packet reordering. The changes to TCP to support FACK are a change in the sender's TCP to use the FACK algorithm for recovery, and, for optimal performance, use of SACK options by the receiver.

In looking for alternative responses to packet corruption, it is noted that TCP segments that are corrupted are often detected at the link level, and are discarded by the link-level drivers. This discard cannot be used to generate an error message to the packet sender, given that the IP header of the packet may itself be corrupted, nor can the discard signal be reliably passed to the receiver, for the same reason. However, despite this unreliability of information, this signaling from the link level to the transport level is precisely the objective here, because, at the TCP protocol level, the sender needs to be aware that the packet loss was not due to network congestion, and that there is no need to take corrective action in terms of TCP congestion behavior.

One approach to provide this signaling from the data link level to the transport level calls for the link-level device to forward a “corruption experienced” Internet Control Message Protocol (ICMP) packet when discarding a corrupted packet[RFC2760]. This approach has the ICMP packet being sent in the forward direction to the receiver, who then has the task of converting this message and the associated lost packet information into a signal to the sender that the duplicate ACKs are the result of corruption, not network congestion. This signal from the receiver to the sender can be embedded in a TCP header option. The sending TCP session will maintain a corruption experienced state for two RTT intervals, retransmitting the lost packets without halving the congestion window size.

As we have noticed, corruption may have occurred in the packet header, and the sender's address may not be reliable. This approach addresses this by having the router keep a cache of recent packet destinations, and when the IP header information is unreliable because of a failed IP header checksum, the router will forward the ICMP message to all destinations in the cache. The potential weakness in this approach is that if network congestion occurs at the same time as packet corruption, the sender will not react to the congestion, and will continue to send into the congestion for a further two RTT intervals. This approach is not without some deployment concerns. It calls for modification to the wireless routers and to the receiver's link-level drivers to generate the ICMP corruption experienced messages, modification to the receiver's IP stack in order to take signals from the IP ICMP processor and from the link level driver and convert them to TCP corruption loss signals within the TCP header of the duplicate ACKs, and modifications to the TCP processor at the sender to undertake corruption-experienced packet loss recovery. Even with these caveats in mind, this approach of explicit corruption signaling is a very promising approach to addressing performance issues with TCP over wireless.

Of course high levels of bit errors is not the only problem facing TCP over wireless systems. Mobile wireless systems are typically small handsets or personal digital assistants, and the application transactions are often modified to reduce the amount of data transferred, given that a limited amount of data can be displayed on the device. In this case, the ratio between payload and IP and TCP headers starts to become an issue, and some consideration of header compression is necessary. Header compression techniques typically take the form of stripping out those fields of the header that do not vary on a packet-by-packet basis, or that vary by amounts that can be derived from other parts of the header, and then transmitting the delta values of those fields that are varying.

Although such header compression schemes can be highly efficient in operation, the limitation of such schemes is that the receiver needs to have successfully received and decompressed the previous packet before the receiver can decompress the next packet in the TCP stream. In the face of high levels of bit error corruption, such systems do introduce additional latencies into the data transfer, and multiple packet drops are difficult to detect and signal via SACK in this case. A more subtle aspect of mobile wireless is that of temporary link outages. For example, a mobile user may enter an area of no signal coverage for a period of time, and attempt to resume the data stream when signal is obtained again. In the same way that there is no accepted way of a link-level driver informing TCP of packet loss due to corruption, there is no way a link-level driver can inform TCP of a link-level outage. In the face of such link-level outages, TCP will assume network level congestion, and in the absence of duplicate ACKs, TCP retransmission timers will trigger. TCP will then attempt to restart the session in slow-start mode, commencing with the first dropped packet. Each attempt to send the packet will result in TCP extending its retransmission timer using an exponential backoff on each attempt, so that successive probes are less and less frequent. Because the link level cannot inform the sender on the resumption of the link, TCP may wait some considerable time before responding to link restoration. The intention is for the link level to be able to inform the TCP for resumption of the connection following a link outage. One approach is for the link level to retain a packet from each TCP stream that attempted to use the link. When the link becomes operational again, the link-level driver immediately transmits these packets on the link. The result is that the receiver will then generate a response that will then trigger the sender into transmission within a RTT interval. Only a single packet per active TCP stream is necessary to trigger this response, so that the link level does not need to hold an extensive buffer of undeliverable packets during a link outage. Of course if the routing level repaired the link outage in the meantime, the delivery of an out-of-order TCP packet would normally be discarded by the sender.

The bottom line here is the question: Is TCP suitable for the mobile wireless environment? The answer appears to be that TCP can be made to work as efficiently as any other transport protocol for the mobile wireless environment. However, this does imply that some changes in the operation of TCP need to be undertaken, specifically relating to the signaling of link-level states into the TCP session and use of advanced congestion control and corruption signaling within the TCP session.

Although it is difficult to conceive of a change to every TCP stack within the deployed Internet to achieve this added functionality, there does exist a middle ground between the “walled garden” approach and open IP. In this middle ground, the wireless systems would have access to “middleware,” such as Web proxies and mail agents. These proxies would use a set of TCP options when communicating with mobile wireless clients that would make the application operate as efficiently as possible, while still permitting the mobile device transparent access to the Internet for other transactions.

Unbundling TCP — Stream Control Transmission Protocol

There are occasions where the application finds the control functions of TCP too limiting. In the case of handling Public Switched Telephone Network (PSTN) signaling across an Internet network, the application requirements are somewhat different from those of TCP delivered service. PSTN signaling reliable delivery is important, but the individual transactions within the application are included within each packet, so the concept of preservation of strict order of delivery is unnecessary. Relaxation of this requirement of strict order of packet delivery allows the transport protocol to function more efficiently, because there is no head-of-line blocking at the receiver when awaiting retransmission of lost packets. TCP also assumes the transfer of a stream of data, so that applications that wish to add some form of record delineation to the data stream have to add their own structure to the data stream. In addition, the limited scope of TCP sockets complicates the support of a high availability application that may use multi-homed hosts, and TCP itself is vulnerable to many attacks, such as SYN attacks. The intention of the Stream Control Transmission Protocol (SCTP) is to address these application requirements[RFC1144].

The first major difference between SCTP and TCP occurs during initialization, where the SCTP endpoints exchange a list of SCTP endpoint addresses (IP addresses and port numbers) that are to be associated with the SCTP session. Any pair of these source and destination addresses can be used within the SCTP session. The startup of SCTP is also altered into a four-way handshake, where the initiator sends a tag value to the other end, which then responds with a copy of this tag and a tag of its own. At this stage the recipient does not allocate any resources for the connection, making the initialization sequence more robust in the face of TCP SYN-styled attacks. The initiator can then respond to this with an echo of the recipient’s tag (COOKIE-ECHO), and can also attach data to the response, allowing data to be transferred as early as possible in the handshake process.

After the recipient ACKs this message, the SCTP session is now established. The closing of an SCTP session is also different from TCP. In TCP, one side can close its sending function via a FIN TCP packet, and continue to receive packets, operating in a “half-open” state. In SCTP, a close from one side will cause the other end to drain its send queues and also shut down.

SCTP also functions in a form of transport-level multiplexing, where numerous logical streams can be supported across a single transport-level association. Although message order within an individual stream is preserved by SCTP, retransmission within one stream does not impact the operation of any other stream that is supported across the same SCTP transport association. Each stream has an explicit identification and a per-stream sequence identification to support this function. SCTP also provides for non-sequenced message delivery, where a message within a stream is marked for immediate delivery, irrespective of the relative order of the message within a stream (Figure 4).

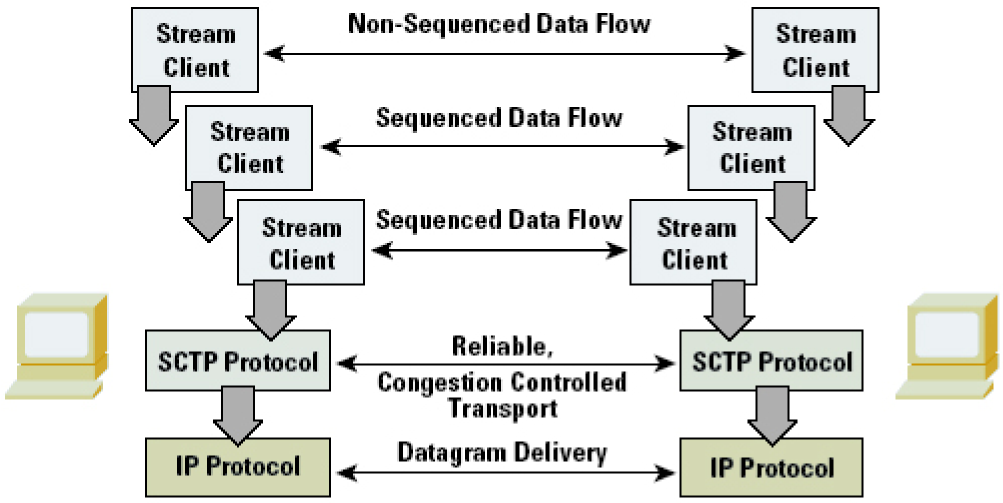

Figure 4: The SCTP Transport Service Model

SCTP explicitly uncouples transport-level reliability and congestion control from per-stream sequenced delivery through the use of a separate transport-level interaction. The transport-level data and ACKs and the corresponding transport-level congestion window controls operate using a transport-level sequence space. This sequence space counts transport-level messages, not byte offsets within the message, so that no explicit window scaling option is necessary for SCTP. The congestion control functions reference those of TCP with fast retransmit and fast recovery, with an explicit specification of the SACK protocol and specification of the maintenance of the transmission timers and congestion control. SCTP also requires the use of MTU path discovery, so that larger transactions will use SCTP-level segmentation, avoiding the IP retransmission problem with lost fragments of a fragmented IP packet. SCTP does use a modified retransmission mechanism to that of TCP. Like TCP, SCTP associates a retransmission timer with each message, and if the timer expires the message is retransmitted and SCTP collapses the congestion window to a single message size. The SCTP receiver will generate SACK reports for a minimum of every second received packet. If a message is within a SACK gap, then after three further such SACK messages, the sender will immediately send the missing messages, and half its congestion window, analogous to the fast retransmit and fast recovery of TCP.

The use of multiple endpoint addresses assumes that each of the endpoint addresses is associated with the same end host, but with a potentially different network path between the two endpoints. SCTP refreshes path availability to each of the endpoint addresses with a periodic keepalive, so that in the event of primary path failure, SCTP can continue by using one of the secondary endpoint addresses.

One could describe SCTP as being overly inclusive in terms of its architecture, and there is certainly a lot of capability in the protocol that is not contained within TCP. The essential feature of the protocol is to use a single transport congestion state between two systems to allow a variety of applications to attach as stream clients. In itself, this is analogous to TCP multiplexing. It also implicitly assumes that every stream is provided the same service level by the network, an assumption shared by almost all transport multiplexing systems. The essential alteration with SCTP is the use of many transport modes: reliable sequenced message streams, reliable sequenced streams with interrupt message capability, and reliable non-sequenced streams. It remains to be seen whether the utility provided by this protocol will become widely deployed within the Internet environment, or whether it will act as a catalyst for further evolution of transport service protocols.

Sharing TCP information — Endpoint Congestion Management

The notion of sharing a single TCP congestion state across multiple reliable streams is one that may also be applied to a mix of reliable and non-reliable data streams that operate concurrently between a pair of endpoints. It is this form of the multiplexing service model that is explored by the congestion manager model. The Congestion Manager is an end-system module that allows a collection of concurrent streams from the host to a single destination to share a common congestion control function, and permits various forms of reliable and non-reliable streams to use the network in a way that cooperates with concurrent congestion controlled flows [RFC 3124].

One of the major motivations for the congestion manager is the observation that the most critical part of network performance management is that of managing the interaction between congestion-controlled TCP streams and non-responsive UDP data streams. In the extreme cases of this interaction, either traffic class can effectively deny service to the other by placing sufficient pressure on the network queuing resources that starve the other traffic class of any usable throughput. The observation made in the motivation for the congestion manager is that applications such as the Web typically open up a set of parallel connections to provide service, sending a mix of reliable flow-controlled data along one connection and unreliable real-time streaming content along another. If the set of flows used a common congestion-control function at the sending host, the collection of flows would utilize the network resources in a manner analogous to a single TCP connection.

The manner of providing this common congestion control function is an advisory function to applications, as shown in Figure 5. One mechanism is that of a callback, where an application inserts a request to send a single message segment with the congestion manager. The Congestion Manager responds with invoking a callback to the requestor when the application may pass the data segment to the protocol driver. The other supported mechanism is that of synchronous transmission, where the Congestion Manager has a callback function that updates the application with a maximal available bit rate, the smoothed round-trip time estimate, and the smoothed linear deviation in the round-trip time estimate. In this mode the application can request further notification only when the network state changes by some threshold amount.

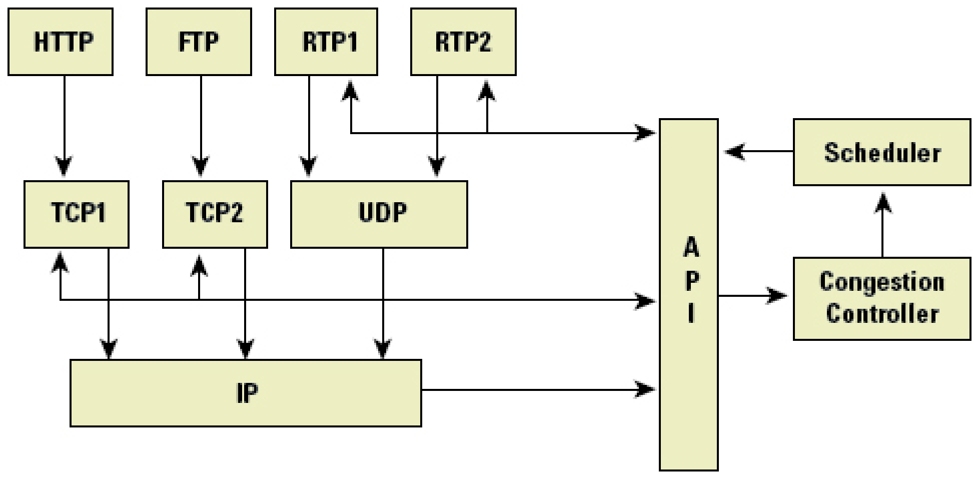

Figure 5: The CM Model, (after [RFC 3124])

For the Congestion Manager to maintain a current picture of the congestion state of the path to the destination, each active stream needs to update the congestion manager as to the response from the remote host. It does this by informing the congestion manager of the number of bytes received, the number of bytes lost, and the RTT measurement, as measured at the application level. The application is also expected to provide an indication of the nature of the loss, as a timeout expiry, a transient network condition, or based on the reception of an ECN signal.

There has been little practical experience as yet with this model of shared congestion control within the Internet environment. There also remains a number of issues about how network performance information is passed back from the receiver to the sender in the absence of an active concurrent TCP session. The concurrent operation of a TCP session with a UDP streaming session to the same destination allows Congestion Manager to use the TCP congestion state to determine the sending capability of the streaming flow.

If the TCP session is idle, or if there is no TCP session, then the UDP streaming application will require some form of receiver feedback. The feedback will need to report on the span of data covered by the report, and the data loss rates and jitter levels, allowing the sender to assess the current quality and capacity of the network path.

This approach, and that of SCTP, are both illustrative of the approach of unbundling the elements of TCP and allowing applications to use combinations of these elements in ways that differ from the conventional monolithic transport-level protocol stack, with the intention of allowing the TCP congestion control behavior to be applied to a wider family of applications.

Datagram Congestion Control Protocol

A similar motivation applies to the Datagram Congestion Control Protocol (DCCP) [DCCP]. This effort is intended to provide TCP congestion control to a single datagram packet flow, rather than the congestion manager's approach of sharing the congestion state of a TCP session with concurrent UDP streams. The interesting aspect of this work is the decision to provide support for two congestion control mechanisms: one acts in a manner similar to TCP, that permits higher throughput but has the potential to operate with rapid rate changes, and the other using a rate control process [RFC3448] that operates at a steadier rate. Both control algorithms support the option of interaction with Explicit Congestion Notification (ECN) [ECN].

The approach taken with DCCP is that of a new transport protocol that, like TCP, uses in-band signaling, and, unlike UDP, uses a bidirectional packet flow of data packets. TCP is a data stream protocol where sequence numbers identify byte positions within the data stream, while DCCP is a packet protocol, using packet sequencing and packet rate control. DCCP defines a 'half-connection', which consists of a congestion-controlled unidirectional data flow and a reverse flow of acknowledgements. DCCP also supports a bi-directional data flow, but this is supported as two semi-autonomous half-connections, where each half-connection uses its own congestion control state, and may even use different congestion control algorithms for each streamed flow. The essential difference to UDP is that this approach places the responsibility for rate control of a streaming flow within the transport layer of the protocol stack, rather than having the application create its own congestion control state based on incorporating information from a reverse flow feedback report into the rate control of the forward flow packet stream.

It may be useful to think of DCCP either as TCP minus byte stream semantics and reliability, or as UDP plus congestion control, handshakes, and acknowledgements. [DCCP]

Unlike TCP, DCCP appears to be leading to a richer model of interaction with the upper level protocol. A TCP socket is one where the application can assume that the data, having been passed into the transport session, will be reliably delivered to the receiver without further application direction, and the transfer rate is a matter for the transport protocol to determine and manage, not the application. Packet streaming applications have traditionally been designed along similar lines, where the application simply passes the data to the UDP transport, often without regards to the currently available network resources. The issue that DCCP attempts to address is that the associated UDP transport is similarly insensitive to prevailing network path characteristics. However, it may not be enough change a streaming application to use a congestion control flow transport. It appears that there is a need for a richer set of feedback options coming from the DCCP transport protocol back to the application, allowing for the application to vary its data encoding to suit the currently available flow bandwidth and current packet loss rate on the network path. The constraint here is that the rate control is on the basis of a sending packet rate. Applications that use a technique of varying packet sizes would need to use a comparable technique that undertakes congestion control based on bit rate control.

There is much more to DCCP, as it has attempted to incorporate much of the more recent experience in TCP modifications that it attempts to guard the session against various forms of attack, attempts to allow the protocol to operate across various forms of middleware, and supports the inclusion of a session identity in order to allow for a form of location address mobility. This brief overview can only offer an introduction to the essential features of this new transport protocol, so further reading on DCCP as another evolutionary development of TCP is well merited.

TCP Evolution

The evolution of TCP is a careful balance between innovation and considered constraint. The evolution of TCP must avoid making radical changes that may stress the deployed network into congestion collapse, and also must avoid a congestion control “arms race” among competing protocols[RFC 2914]. The Internet architecture to date has been able to achieve new benchmarks of network efficiency, and translate this carriage efficiency into ground-breaking benchmark prices for IP-based carriage services. Much of the credit for this must go to the operation of TCP, which manages to work at that point of delicate balance between self-optimization and cooperative behavior.

Widespread deployment of transport protocols that take a more aggressive position on self-optimization will ultimately lead to situations of congestion collapse, while widespread deployment of more conservative transport protocols may well lead to lower jitter and lower packet retransmission rates, but at a cost of considerably lower network efficiency.

The challenges faced with the evolution of TCP is to maintain a coherent control architecture that has consistent behavior within the network, consistent interaction with instances of data flows that use the same control architecture, and yet be adequately flexible to adapt to differing network characteristics and differing application profiles. It is highly likely that we will see continued innovation within Internet transport protocols, but the bounds of such effort are already well recognized.

We can now state relatively clearly what levels of innovation are tolerable within an Internet network model that achieves its efficiency not through enforcement of rigidly enforced rules of sharing of the network resource, but through a process of trust between competing user demands, where each demand is attempting to equilibrate its requirements against a finite network capacity. This is the essence of the TCP protocol.

Appendix: TCP Performance Models

This appendix is an extract from “Advice for Internet Subnet Designers,” [PILC].

TCP Performance Characteristics

Caveat:

Here we present a current "state-of-the-art" understanding of TCP performance. This analysis attempts to characterize the performance of TCP connections over links of varying characteristics.

Link designers may wish to use the techniques in this section to predict what performance TCP/IP may achieve over a new link-layer design. Such analysis is encouraged. Because this is a relatively new analysis, and the theory is based on single-stream TCP connections under "ideal" conditions, it should be recognized that the results of such analysis may differ from actual performance in the Internet. That being said, we have done the best we can to provide information which will help designers get an accurate picture of the capabilities and limitations of TCP under various conditions.

The Formulae

The performance of TCP's AIMD Congestion Avoidance algorithm has been extensively analyzed. The current best formula for the performance of the specific algorithms used by Reno TCP (i.e., the TCP specified in [RFC2581]) is given by Padhye, et al [THROUGHPUT]. This formula is:

$$BW = \frac{MSS}{RTT*sqrt(1.33*p) + RTO*p*[1+32*p^2]*min[1,3*sqrt(.75*p)]}$$

Where:

BW	is the maximum TCP throughout achievable by an individual TCP flow
MSS	is the TCP segment size being used by the connection
RTT	is the end-to-end round trip time of the TCP connection
RTO	is the packet timeout (based on RTT)
p	is the packet loss rate for the path (i.e. .01 if there is 1% packet loss)

Note that the speed of the links making up the Internet path does not explicitly appear in this formula. Attempting to send faster than the slowest link in the path causes the queue to grow at the transmitter driving the bottleneck. This increases the RTT, which in turn reduces the achievable throughput.

This is currently considered to be the best approximate formula for Reno TCP performance. A further simplification to this formula is generally made by assuming that RTO is approximately 5*RTT.

TCP is constantly being improved. A simpler formula, which gives an upper bound on the performance of any AIMD algorithm which is likely to be implemented in TCP in the future, was derived by Ott, et al [IDEAL].

$$BW = C \quad \frac{MSS}{RTT} \frac{1}{sqrt(p)}$$

where C is 0.93.

Assumptions

Both formulae assume that the TCP Receiver Window is not limiting the performance of the connection. Because the receiver window is entirely determined by end-hosts, we assume that hosts will maximize the announced receiver window to maximize their network performance.

Both of these formulae allow BW to become infinite if there is no loss. However, an Internet path will drop packets at bottleneck queues if the load is too high. Thus, a completely lossless TCP/IP network can never occur (unless the network is being underutilized).

The RTT used is the arithmetic average, including queuing delays.

The formulae are for a single TCP connection. If a path carries many TCP connections, each will follow the formulae above independently.

The formulae assume long-running TCP connections. For connections that are extremely short (<10 packets) and don't lose any packets, performance is driven by the TCP slow-start algorithm. For connections of medium length, where on average only a few segments are lost, single connection performance will actually be slightly better than given by the formulae above.

The difference between the simple and complex formulae above is that the complex formula includes the effects of TCP retransmission timeouts. For very low levels of packet loss (significantly less than 1%), timeouts are unlikely to occur, and the formulae lead to very similar results. At higher packet losses (1% and above), the complex formula gives a more accurate estimate of performance (which will always be significantly lower than the result from the simple formula).

Note that these formulae break down as p approaches 100%.

References and Further Reading

There is a large volume of papers that have been published on TCP and its evolution over the years, and the collection of references provided here refer only to the documents that have been directly referenced in this document. TCP represents one of the more fascinating aspects of the entire Internet Protocol suite, and even after many years of intense study there is still much more to learn about the behavior of this protocol.

[RFC 1144] Jacobson, V., *Compressing TCP/IP Headers for Low-Speed Serial Links*, RFC 1144, February 1990.

[RFC 1191] Mogul, J., Deering. S., *Path MTU Discovery*, RFC 1191, November 1990.

[RFC 1323] Jacobson, V., Braden, R., Borman, C., *TCP Extensions for High Performance*, RFC 1323, May 1992.

[RFC 1644] Braden, R., *T/TCP—TCP Extensions for Transactions Functional Specification*, RFC 1644, July 1994.

[RFC 2018] Mathis, M., Mahdavi, J., Floyd, S., Romanow, A., *TCP Selective Acknowledgement Options*, RFC 2018, October 1996.

[RFC 2448] Allman, M., Glover, D., Sanchez, L., *Enhancing TCP over Satellite Channels Using Standard Mechanisms*, RFC 2488, January 1999.

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

[RFC 2508] Casner, S., Jacobson, V., *Compressing IP/UDP/RTP Headers for Low- Speed Serial Links*, RFC 2508, February 1999.

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

[RFC 2760] Allman, M., editor, *Ongoing TCP Research Related to Satellites*, RFC 2760, February 2000.

[RFC 2883] Floyd, S., Mahdavi, J., Mathis, M., Podolsky M., *An Extension to the Selective Acknowledgement (SACK) Option for TCP*, RFC 2883, July 2000.

[RFC 2914] Floyd, S., editor, *Congestion Control Principles*, RFC 2914, September 2000.

[RFC 2960] Stewart, R., et al., *Stream Control Transmission Protocol*, RFC 2960, October 2000.

[RFC 3042] Allman, M., Balakrishnan, H., Floyd, S., *Enhancing TCP's Loss Recovery Using Limited Transmit*, RFC 3042, January 3042.

[RFC 3124] Balakrishnan, H., Seshan, S., *The Congestion Manager*, July 2000.

[RFC 3465] Allman, M., *TCP Congestion Control with Appropriate Byte Counting*, RFC 3465, February 2003.

[BEHAVIOUR] M. Mathis, M., Semke, J., Mahdavi, J., Ott, T., *The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm*, Computer Communication Review, Vol. 27, No. 3, July 1997.

[CTL] Jacobson, V., *Congestion Avoidance and Control*, ACM SIGCOMM, 1988.

[E2ECTL] Floyd, S., Fall, K., *Promoting the Use of End-to-End Congestion Control in the Internet*, Submitted to IEEE Transactions on Networking.

[IDEAL] Ott, T., Kemperman, J., Mathis, M., *The Stationary Behavior of Ideal TCP Congestion Avoidance*, available at: ftp://ftp.bellcore.com/pub/tjo/TCPwindow.ps

[MEASURE] Claffy, K., Miller, G., Thompson, K., *The Nature of the Beast: Recent Traffic Measurements from an Internet Backbone*, INET'98 Proceedings, Internet Society, July 1998. Available at: http://www.isoc.org/inet98/proceedings/6g/6g_3.htm

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

[PILC] Karn, P., Falk, A., Touch, J., Montpetit, M., Mahdavi, J., Montenegro, G., Grossman, D., Fairhurst, G., *Advice for Internet Subnet Designers*, work in progress (draft-ietf-pilc-link-design-15, December 2003.

[TCP] Postel, J., *Transmission Control Protocol*, RFC 793, September 1981.

[THROUGHPUT] Padhye, J., Firoiu, V., Towsley, D., Kurose, J., *Modeling TCP Throughput: A Simple Model and Its Empirical Validation*, UMASS CMPSCI Tech Report TR98-008, Feb. 1998.

Geoff Huston

Disclaimer

The above views do not represent the views of the Internet Society. They were possibly the opinions of the author at the time of writing this article, but things always change, including the author's opinions!

About the Author

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. He is the Executive Director of the Internet Architecture Board, and a member of the Board of the Public Interest Registry. He was an inaugural Trustee of the Internet Society, and served as Secretary of the Board of Trustees from 1993 until 2001, with a term of service as chair of the Board of Trustees in 1999 and 2000. He is author of a number of Internet-related books.