Geoff Huston
October 2015

Transport Protocols

One of the early refinements in the Internet protocol model was the splitting of the original Internet protocol from a single monolithic specification [1] into the Internet Protocol (IP) and a pair of transport protocols. The Internet Protocol layer is intended to be used by the internal switches within the network to forward the packet to its intended destination, while the Transport Protocol layer is intended to be used by the source and destination systems. In this article I'd like to look at what we've been doing with these transport protocols.

There are two end-to-end transport protocols in common use in today's Internet: the User Datagram Protocol (UDP) [2] and Transmission Control Protocol (TCP) [3]. Why just two? Surely we've thought of many ways to control the flow of data across a packet switched network, and in an architecture as open as the Internet it seems reasonable to ask why haven't see seen the emergence of a plethora of end-to-end transport protocols. There have been no shortage of attempts, and on any Unix-like system an inspection of `/etc/protocols` provides a list of 130 such protocols that are layered on top of IP. Now some of these are special purpose protocols, such as protocol 89, used by the OSPF routing protocols, and others are encapsulation protocols, such as protocol 41 for IPv6-in-IPv4 encapsulation. But there are a fair number of end-to-end protocols in that list as well, such as protocol 27, the Reliable Datagram Protocol (RDP), of protocol 132, the Stream Control Protocol (SCTP). But most of these end-to-end transport protocols find it hard to survive in the public Internet. The issue today is that our middleware has got the better of us.

Much of the Internet shelters behind all kinds of middleware. These can take the form of firewalls and filters, that work on the principle of explicit permission rather than exception. These units will commonly permit protocols 6 and 17 (TCP and UDP), but as far as permitted end-to-end protocols go, that's normally it. It's not just the paranoia of firewalls. We have load balancers, packet shapers, and many other forms of intercepting middleware that can recognise and manipulate TCP sessions, but they conventionally don't do much else. So, for all practical purposes the Internet only allows the end-to-end transport protocols TCP and UDP. So lets look at these two protocols in a little more detail.

UDP

UDP is a very simple abstraction of the basic IP datagram, in that like IP itself, UDP is an unreliable medium. Mirroring the service characteristics of IP, Packets sent using UDP may or may not be received by their intended destination. UDP packets may be reordered, duplicated or lost. There is no flow control and no throttling in UDP. The packet quantization in UDP is explicit: if data is split into two UDP packets by the sender, then the receiver will collect the data using two distinct read operations.

UDP is intended to be used for extremely simple transactions that do not require session context. The Domain Name System (DNS) and the Network Time Protocol (NTP) are good examples of applications that make use of UDP to support a very efficient query / response transaction model. Typically a sender generates a query packet with space for an answer and the response is the same packet with the answer field completed.

These days UDP is regarded far more cautiously. The lack of a session context implies that most transactions are unencrypted, and are not only readily tapped by third parties, but they are highly susceptible to passing off attacks, where someone other than the intended recipient generates a response, possibly intending to mislead or dupe the original querier. For example, DNS interceptors have become a very common means of performing content filtering on today's internet. More insidiously, UDP is a common platform for mounting various forms of denial of service attacks. Many UDP servers, such as authoritative DNS servers, need to answer all UDP queries, so they are not in a position to only answer certain "genuine" queries. All queries tend to look genuine in UDP. This has been exploited by attackers who place the IP address of the intended victim in the source address of the UDP packet. With enough queries sent to enough servers all with the same source address, its possible to enrol UDP servers into being unwitting attackers. So today UDP has fallen from grace.

If its not UDP then its likely to be TCP.

TCP

TCP is a reliable end-to-end flow controlled stream protocol. A stream of data passed into a TCP socket at one end will be read as a stream of data at the other end. As it's a stream protocol, the packet quantisation is hidden from the application, as are the mechanics of flow control, loss detection and retransmission, session establishment and session teardown. TCP will not preserve any inherent timing within the data stream, but will preserve the integrity of the stream.

Within any packet-switched network when there is contention for a common output port, or when sustained packet volumes exceeds the 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. One approach is to have each pair of switches conduct a reliable protocol over their common link, and perform this detection and correction of packet loss on a hop-by-hop basis. TCP uses a different approach and makes no assumptions about the reliability of each hop. Instead TCP use an end-to-end data sequence numbering between the two communicating systems that allows them to identify their context within the stream. When an in-sequence packet is received by a host it will send back as a positive acknowledgement (ACK) to the sender the sequence number of the final byte in the packet. This ACK is a "cumulative ACK" in that it is saying that all data in the stream up to the sequence number contained in the ACK has been received. There are no negative acknowledgements (NACKs) in TCP. When an out-of-order packet arrives, the receiver will also send an ACK, but it will contain the sequence number of the highest in-order byte that has been received. This form of reliable protocol design is termed "end-to-end" control, as distinct to "hop-by-hop" control, as TCP does not assume that interior switches will attempt to correct packet drops at a hop-by-hop level.

> Operating TCP over a reliable hop-by-hop controlled environment is suboptimal. The early experience of running TCP over these hop-by-hop reliable protocols such as X.25 packet switched systems, and later ARQ reliable cellular switched radio systems, pointed to significant problems with this approach in terms of the efficiency of TCP. As we will see, TCP makes use of timing signals to guide its behaviour, and when each link in the path is performing a "stop and repair" form of reliable hop-by-hop control, then this will impose jitter (variation in the time to send a packet and receive its acknowledgement) on the packet flow. This imposed jitter will disrupt TCP's internal estimate of the Round Trip Time, which will cause its flow control processing to behave erratically.

TCP uses these ACKs as a form of feedback from the receiver back to the sender in order 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 it can send data into the network at the same rate as it is leaving the network at its 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 data into the network at the same rate as data is removed from the network at the other end.

Critically, today's Internet assumes that most of the network's resources are devoted to passing TCP traffic, and it also assumes that the flow control algorithms used by these TCP sessions all behave in approximately similar ways. If the switching and transmission resources of the network are seen as a common resource, then the assumption about the uniform behaviour of TCP sessions implies that these end-to-end transport sessions will behave similarly under contention. The result is that, to a reasonable level of approximation, a set of concurrent TCP sessions will self-equilibrate to give each TCP session an equal share of the common resource. In other words the network itself does not have to impose "fairness" on the TCP flows that pass across it – as long as all the flows are controlled by a uniform flow control algorithm then the flows will interact with each other in a manner that is likely to allocate an equal proportion of the network's resources to each active TCP flow. At least that's the theory.

But do theory and practice align? Is this the case in today's Internet and what may be changing with these assumptions about TCP behaviour?

TCP Flow Control – TCP Tahoe and TCP Reno

Perhaps surprisingly, TCP does not have a single flow control algorithm. While the common TCP protocol specification defines how to establish and shutdown a session, and defines the way in which received data is acknowledged back to the sender, the core protocol specification for TCP does not specify how the two ends negotiate the speed at which data is passed between them. A simple approach is to send data until the sender's buffer of unacknowledged sent data is full. Received ACKs will allow the sender to shrink this buffer, and it can then send further data into the network until the buffer is buffer is again full.

However, this simple approach has its problems when used by a number of simultaneous sessions, leading to what as observed at the time as "congestion collapse". The TCP sessions interact in a way that causes large scale packet drop, and the loss of ACK signalling causes all senders to retransmit, and so on. A study of this behaviour in the 1980's lead to the introduction of a "flow control" approach to TCP behaviour [4].

One of the early TCP flow control algorithms was TCP Tahoe, first used in the 4.3BSD operating system. In this flow control framework there are two distinct phases of behaviour: the "Slow Start" phase, where the sending rate is doubled every round trip time (RTT) interval, and a "Congestion Avoidance" phase, where the sending rate is increased by a fixed amount (one Message Segment Size (MSS)) in each RTT interval [5].

Slow Start phase starts the data flow in a very conservative manner, sending just one packet into the network and awaiting its corresponding ACK. However, each received ACK causes the sender to double its sending window size, so that the sender will successively send 2, 4, 8 and so on packets into the network upon each RTT interval. This generates an exponentially increasing data rate into the network, and the TCP session will rapidly either reach the maximum transmission receiving rate of the remote receiver, the maximum initial transmission rate of the sender (the "slow start threshold", or *ssthresh*), or it will push the network into congestion to the point of packet loss. In the first cases the TCP flow rate will stabilise at the receiver's maximum rate. In the second case, where the sending rate exceeds a local threshold the sender will then move into its Congestion Avoidance mode, and continue

to increase the sending rate until either the receiver's maximum rate is reached, or until packet loss occurs.

If an ACK for a packet fails to arrive within the RTT interval then TCP Tahoe assumes that the sending packet has been lost in the network. This packet loss event causes the local *ssthresh* value to be set to half of the current sending rate, and the sending rate is set back to a single packet and slow start is resumed. An idealised picture of the resultant behaviour of a TCP Tahoe-controlled data flow is shown in Figure 1. The idea is that at startup the sender rapidly probes upward in speed to determine the upper bound on the sustainable data rate. At that point the sender now has an approximate rate of rate. The reasoning is that it assumes that congestion loss occurred within the last RTT interval then the maximal sustainable sending rate is somewhere between half of the sending rate of the last RTT interval and the sending rate of the last RTT interval. Tahoe then performs slow start again, but stops at this halved rate, which it found from the previous slowstart phase was sustainable by the network. The sender enters into congestion avoidance mode, and probes far more carefully into the next range of sending speeds, increasing the sending rate by a single segment each RTT interval. Again, Tahoe will assume that it has overstepped the mark when packet loss occurs , and again it will set *ssthresh* to half the sending rate and restart the flow in slow start.

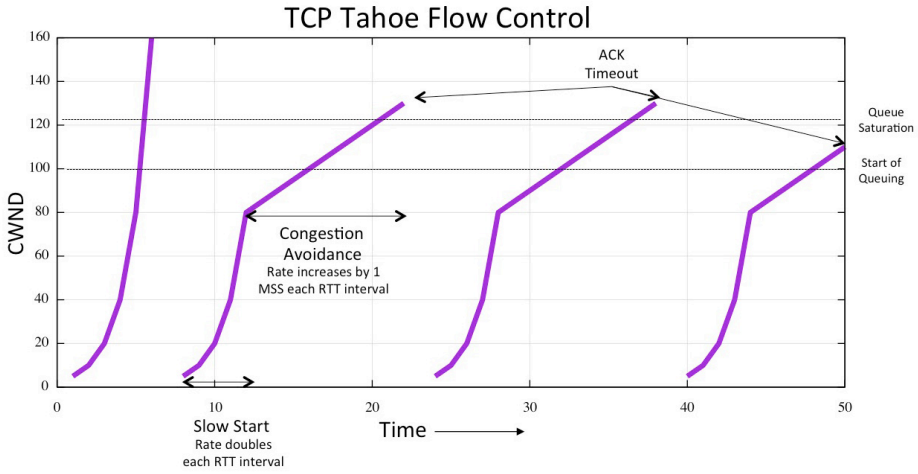

Figure 1 – Idealised TCP Tahoe Flow Control

Tahoe uses the lack of an ACK packet to signal congestion loss. Tahoe carries an internal estimate of the RTT, and when the ACK has not been received at the RTT interval (plus some interval to allow for jitter), then Tahoe will perform its reset to slow start. This response to packet loss 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.

To address this, TCP Reno 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 last in-sequenced byte. In this mode the sender immediately retransmits the lost packet and then continues with ACK pacing while duplicate ACKs continue to arrive. Once it receives an indication that the recovery transmissions have been received (by the ACK counter moving past the sequence number of the lost data), the sender then resumes congestion avoidance, with a rate equal to one half of the rate used when the duplicate ACKs were received.

The protocol reacts sharply to these duplicate ACK signals of network congestion, but only to one half of the previous sending rate, and then resumes gradually increasing its sending rate in order to equilibrate with concurrent TCP sessions. If it fails to recover the missing packets using this fast

recovery mechanism then it collapses its sending window back to 1 and re-enters Slow Start mode. Figure 2 show the idealised behaviour of TCP Reno.

The intent in the Congestion Avoidance mode is for the sender to carefully probe into the point where the network path is congested, gradually increasing its data flow pressure on the network. In Congestion Avoidance mode the duplicate ACKs will cause the sender to halve its sending rate, attempt to recover from the lost packet, and if successful then continue in this congestion avoidance mode from the new sending rate.

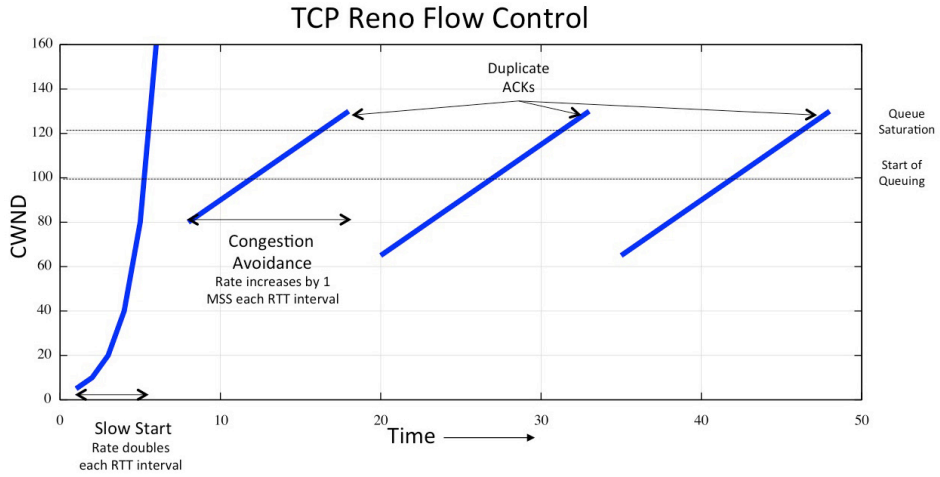

Figure 2 – Idealised TCP Reno Flow Control

In steady state the TCP Reno Congestion Avoidance algorithm ideally avoids restarting the session using slow start, and instead tries to continue the flow with the same assumptions about the likely onset of congestion loss. This is a process of Additive Increase in the sending window (by one segment each RTT) and Multiplicative Decrease in the sending window (by halving its size) at the onset of congestion, or AIMD.

TCP Reno's AIMD algorithm tends to place high levels of pressure on the buffers in the network while there is still available buffer space, and react less dramatically when the buffers eventually overfill and reach the packet drop point. TCP Reno's flow control is far more efficient than its Tahoe predecessor, but this is still a "boom and bust" form of feedback control, which attempts to drive the network into congestion overload and then backoff to allow the buffers to drain. Despite these shortcomings Reno has been the mainstay of the Internet for more than a couple of decades, and it remains the benchmark against which other TCP flow control algorithms are compared.

Better than Reno

There have been a number of reasons to break out of Reno's form of TCP flow control behaviour. One approach is to use a more even packet flow across the network, and remove some of the "jerkiness" inherent in TCP Reno. There is also the suspicion that a more "sensitive" flow control application could achieve a superior outcome than TCP Reno. In other words, a different TCP flow control algorithm could achieve better than its "fair share" when competing against a set of concurrent TCP Reno flows!

The first of these approaches we will look at here is a simple change. In an attempt to double the pressure on other concurrent TCP sessions the AIMD algorithm can be adjusted by increasing the sending speed a larger constant amount each RTT, and decreasing it by less following packet loss. Tthis approach is used by MulTCP. For example, if the sending rate was increased by 2 maximum segment size (MSS) units each Round Trip Time (RTT) instead of 1 MSS in TCP Reno, and the sending rate was reduced by one quarter rather than one half upon receipt of a duplicate ACK, then the resultant behaviour would, in an approximate sense, behave like two concurrent TCP sessions. In a fair sharing

scenario this form of flow control would attempt to secure double the network resources devoted to this session as compared to an equivalent TCP Reno session.

Another variant of this approach is "Highspeed TCP", which increases its frequency of probing into potentially claimable capacity by increasing its sending rate by a larger volume, while keeping its reduction rate at a constant value. So rather than Reno's increase of the congestion window by 1 each RTT, Highspeed TCP uses a calculation for the inflation rate per RTT that rises above 1 as the congestion window grows larger. This protocol will probe for the packet loss onset at a far higher frequency than either TCP Reno or MulTCP, once the sender is operating with a large congestion window and is capable of accelerating to much higher flow speeds in a much shorter time interval.

BIC, and its variant CUBIC, use a non-linear increase function rather than a constant rate increase function. Instead of increasing the speed by a fixed amount each RTT in Congestion Avoidance mode BIC remembers the sending rate at the onset of packet drop, and each RTT increases its speed by one half of the difference between the current sending rate and the assumed bottleneck rate. BIC quickly drives the session towards the bottleneck capacity, and then probes more cautiously once the sending speed is close to the bottleneck capacity. (Figure 3) Again, compared to a Reno flow session BIC should produce a superior outcome.

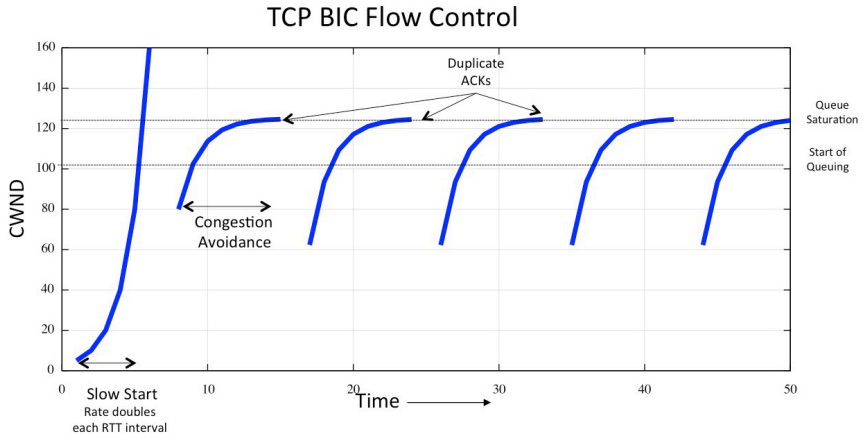

Figure 3 – Idealised TCP BIC Flow Control

Other flow control algorithms move away from using packet loss as the control indication and tend to oscillate more frequently around the point of the onset of queuing in the routers in the network path. This form of feedback control makes the sender sensitive to the relative time differences between sent packets and received ACKs. An example is "packet-pair" flow controlled TCP, where the sending rate is increased as long as the time interval between two packets being sent is equal to the time interval of the received ACKs. If the ACK interval becomes larger then this is interpreted as the onset of queuing in the sending path, and the sending rate is decreased until the ACK timing interval once again equals the send timing interval.

Recent Microsoft systems use Compound TCP, which combines TCP Reno and delay-based flow control. The algorithm attempts to measure the amount of in-flight data held in queues (higher delay traffic) and upon packet loss the algorithm will reduce its sending rate to below the onset of the sender's estimate of growth in queuing.

Apple's Macintosh systems use New Reno, a variant of the Reno flow control algorithm that improves Reno's loss recovery procedure, but is otherwise the same AIMD control algorithm.

Linux kernels have switched to use CUBIC, a variant of the BIC algorithm that uses a cubic function rather than an exponential function to govern window inflation.

There is a delicate balance between the network and the flow control behaviour of TCP, based around the management of the queue buffers in the networks' switches. TCP tends to interpret packet loss as a signal that the bottleneck point in the network path has filled its queue buffers to the point of overflow, and maintaining the same sending rate will only exacerbate packet loss and result in flow inefficiency. The intent of rate halving as a response to congestion loss was to reduce the sending flow rate to a point below the bottleneck capacity, thus allowing the queue to drain, and the slow increase in congestion avoidance was to ensure that this lower flow rate was sustained for long enough to allow the queue buffers to completely drain. The issue here is that a queue that never drains below some minimum behaves in precisely the same was as a delay line coupled with a shorted queue. So the rate halving and the gradual recovery are intended to exercise the full extent of the router's queues and reduce the level of packet loss and transport inefficiency.

The TCP flow algorithms that modify this behaviour tend to work best when used in an environment where all other flows behave more conservatively. In an environment where all the concurrent sessions across a congested link use a RENO-like AIMD behaviour, then a single session that uses a more aggressive response to packet loss, such as CUBIC, will tend to exert a greater pressure on the concurrent TCP RENO-like sessions and gain a greater share of available network resources. Of course this strategy only works when there is a diversity of flow control algorithms in use.

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. There is one fundamental assumption made by these TCP flow control algorithms:

Packet Loss and jitter in the RTT is due to network congestion.

This basic assumption is complemented by a number of additional considerations:

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, RTT instability 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, and that the underlying RTT of a network path is stable.

A best-path route-selection routing 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.

Crossing the Beams: TCP in UDP

Other approaches to the evolution of end-to-end transport have headed further away from conventional TCP and change the behaviour of both the server and the client. One way for an application to do this is to avoid the use of the operating-system provided implementation of TCP completely, and place a TCP-styled reliable flow control streaming protocol into the application data flow, and use the operating system's UDP interface to pass packets to and from the network.

This approach has been used for many years in the BitTorrent streaming application (LEDBAT), and more recently by Google in their experiments with QUIC.

Google's QUIC (Quick UDP Internet Connections) uses a TCP emulation in UDP. QUIC emulates the reliable sliding window protocol used by TCP within a UDP packet flow. QUIC incorporates the functionality of secure transport sessions and slots between a conventional HTTP/2 API as the application interface and UDP (using port 443) as the end-to-end transport protocol.

QUIC implements TCP CUBIC flow control, and also adds to this a number of tweaks to this algorithm.

This includes the use of Selective Acknowledgement (SACK) when enabled by the receiver to detect the case of multiple packet drops in a single RTT window. It also includes an 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, Forward Acknowledgements with Rate Halving (FACK), is 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. The normal 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 implementation also includes Tail Loss Probe (TLP) a technique that responds to ACK timeout by resending the trailing segment, eliciting a SACK response of the missing segments that can then be repaired with FACK. It also supports Forward Retransmission Timeout (F-RTO) to mitigate spurious cases where the sender reverts to slow start following an ACK timeout, and Early Retransmit to support the case of duplicate ACKs received while the sender's congestion window is small to also prevent spurious state changes to slow start.

One thing QUIC can do which is not possible with many TCP tweaks, is play with some of the fundamental mechanisms of TCP, as there is no legacy issue of a modified TCP sender communicating with a conventional unmodified TCP receiver. For example QUIC uses a new sequence number for retransmitted segments, allowing the sender to distinguish between ACKs for the original segment and ACKs for its retransmitted counterpart. QUIC also always uses TLS encryption and plans to adopt TLS 1.3 when that spec is complete. It has already adopted its zero RTT handshake.

Google intend to push this further by using Forward Error Correction (FEC), so that the receiver can repair certain forms of packet low without any retransmission at all, and also add Multipath so as to allow platforms with multiple network interfaces (such as mobile devices) to load share across all active interfaces.

QUIC performs bandwidth estimation as a means of rapidly reaching an efficient sending rate. SPDY further assists QUIC by multiplexing multiple application sessions within a single end-to-end transport

protocol session. This avoids the startup overhead of each TCP session, and leverages the observation that TCP takes some time to establish the network's bottleneck capacity. The use of UDP also avoids intercepting middleware that performs deep packet inspection on TCP flows and modifies their advertised window size to perform external moderation on TCP flow rate.

This is certainly an interesting approach. It breaks out of the issue of backwards compatibility by operating the transport session over UDP, so that there are no legacy TCP considerations that have to be considered.

There is, however, one issue with the use of UDP as a substitute for TCP, and while public reports from Google on this topic have not been published, it is a source of concern. The issue concerns the use of UDP through Network Address Translators (NATs) and the issue of address binding times within the NAT. In TCP a NAT takes its directions from TCP. When the NAT sees an opening TCP handshake packet from the "inside" it creates a temporary address binding and sends the packet to its intended destination (with the translated source address of course). The reception of the response part of the handshake at the NAT causes the NAT to confirm its binding entry and apply it to subsequent packets in this TCP flow. The NAT holds state until it sees a closing exchange or a reset signal that closes the TCP session, or until an idle timer expires. For TCP the NAT is attempting to hold the binding for as long as the TCP session is active. For NATs, UDP is different. Unlike TCP there is no flow status information in UDP. So when the NAT creates a UDP binding it has to hold it for a certain amount of time. There is no clear technical standard here, so implementations vary. Some NATs use very short timers and release the binding quickly, which matches the expectation of the use of UDP as a simple query/response protocol. The use of UDP as an ersatz packet framing protocol for user-level TCP implementation requires the NAT to hold the UDP address binding for longer intervals, corresponding to the hidden TCP session. Some NATs will do so, while others will destroy the binding even though there are still UDP packets active, thus disturbing the hidden TCP session. QUIC assumes that a NAT will hold an idle UDP binding open for 30 seconds. If the NAT is more aggressive that that, then QUIC will fail over to conventional TCP.

This illustrates the level of compromise in today's environment between end-to-end protocols and network middleware. TCP sessions are being modified by active middleware that attempts to govern the TCP flow rate by active modification of window sizes within the TCP session, negating some of the efforts of the TCP session to optimise its flow speed. TCP in UDP passes control of the TCP flow management to the application, and hides the TCP flow parameters from the network. However, UDP sessions are susceptible to interruption by NAT intervention, as some NATs assume that UDP is only used for micro-sessions, and long held UDP sessions are some form of anomalous behaviour that should be filtered by removing the UDP port binding in the NAT.

The End of End-to-End?

Where to now?

It's refreshing to see that the technology of end-to-end protocols is not ossified and static. Our understanding of how to make efficient and effective end-to-end protocols is one that is continually evolving is subtle but important ways.

What is amazing is that TCP has been able to provide an efficient service, whether it's tens of bits per second or billions of bits per second. What is also amazing is that TCP is efficient whether it's the only conversation on a wire, or whether its one of millions of simultaneous TCP conversations on the wire. But what is truly amazing is that this technology, now deployed on billions of devices is still malleable and adaptable. We can still make it better.

End-to-End has definitely not ended yet!

Further Reading

[1] Vinton G. Cerf, Robert E. Kahn, (May 1974). "*A Protocol for Packet Network Intercommunication*" (PDF). *IEEE Transactions on Communications* 22 (5): 637–648. doi:10.1109/tcom.1974.1092259. http://ece.ut.ac.ir/Classpages/F84/PrincipleofNetworkDesign/Papers/CK74.pdf

[2] Postel, J. "*User Datagram* protocol", RFC768, 28 August 1980. https://tools.ietf.org/html/rfc768

[3] Postel, J. "*Transmission Control Protocol*", RFC794, September 1981. https://tools.ietf.org/html/rfc794

[4] V. Jacobsen, M. Karels, Congestion Avoidance and Control, 1988, http://ee.lbl.gov/papers/congavoid.pdf, retrieved 21 June 2015.

[5] W. Stevens, "*TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms*," RFC2001, January 1997. https://tools.ietf.org/html/rfc768

[6] M. Allman, V. Paxon, E. Blanton, TCP Congestion Control, RFC5681, September 2009.

[7] Peter Dodcal, "15 Newer TCP Implementations", http://intronetworks.cs.luc.edu/current/html/newtcps.html, retrieved 21 June 2015.

[8] Fast: https://en.wikipedia.org/wiki/FAST_TCP, retrieved 21 June 2015.

[9] Google's QUIC: https://www.chromium.org/quicfile://localhost/, http/::blog.chromium.org:2015:04:a-quic-update-on-googles-experimental.htm, retrieved 21 June 2015.

[10] IETF activity on TCP flow control: TCP Maintenance and Minor Extensions (tcpm) https://datatracker.ietf.org/wg/tcpm/charter/

Author

Geoff Huston B.Sc., M.Sc., is the Chief Scientist at APNIC, the Regional Internet Registry serving the Asia Pacific region. He has been closely involved with the development of the Internet for many years, particularly within Australia, where he was responsible for building the Internet within the Australian academic and research sector in the early 1990's. He is author of a number of Internet-related books, and was a member of the Internet Architecture Board from 1999 until 2005, and served on the Board of Trustees of the Internet Society from 1992 until 2001 and chaired a number of IETF Working Groups. He has worked as a an Internet researcher, as an ISP systems architect and a network operator at various times.

www.potaroo.net

Disclaimer