Designing DCCP: Congestion Control Without Reliability

Paper #428—14 pages

Abstract

The future of Internet transport is unreliable: fast-growing applications like streaming media and telephony prefer timeliness to reliability, and thus prefer UDP to TCP. Unfortunately, UDP lacks congestion control, so these applications are unsafe for use on underprovisioned best-effort networks unless they implement congestion control themselves—a difficult task. They might, however, happily use *congestion-controlled* unreliable transport. DCCP, the Datagram Congestion Control Protocol, is a new transport protocol in the TCP/UDP family that provides a congestion-controlled flow of unreliable datagrams. It aimed to add to a UDP-like foundation the minimum TCP mechanisms necessary to support congestion control. A simple task, we thought; but removing reliability, and especially cumulative acknowledgements, forced us to reconsider almost every aspect of TCP's design. The resulting protocol sheds light on how congestion control interacts with unreliable transport, how modern network constraints impact protocol design, and, particularly, how TCP's reliable bytestream semantics intertwine with its other mechanisms, including congestion control.

1 Introduction

Providing just the right set of functionality in a network protocol is subtle, and touches on issues of modularity, efficiency, flexibility, and fate-sharing. One of the best examples of getting this right is the split of the original ARPAnet NCP functionality into TCP and IP. We might argue about a few details, such as whether the port numbers should have been in IP rather than TCP, but even with the benefit of 25 years of hindsight, the original functional decomposition still looks remarkably good. The key omission from both TCP and IP was clearly congestion control, which was retrofitted to TCP in 1988 [20]. Protocols other than TCP were appropriately left alone. TCP-like congestion control, which combines congestion control with reliable bytestream semantics, isn't appropriate for all applications; in DNS-like request/response protocols, for example, congestion control must be applied *between* connections, not within them.

However, recent years have seen a large increase in applications using UDP for long-lived flows. These applications, which range from streaming media to Internet telephony, videoconferencing, and games, all share a preference for timeliness over reliability. That is, given a chance to retransmit an old packet or to transmit a new packet, they often choose the new packet—because, by the time the old packet arrived, it would be useless anyway: in media applications, users often prefer bursts of static to choppy rebuffering delay; in games, only the latest position information matters. TCP's reliable bytestream delivery can introduce arbitrary delay, and cannot be told to forget old data, making it inappropriate for these applications. Systems such as Time-lined TCP [27] retrofit some support for time-sensitive data onto TCP, but do so using a specific deadline-based policy. Real applications may have more complex policies that take account, for example, of different levels of importance and interdependencies between various application-level messages—the canonical example being MPEG's key frames (I-frames) and incremental frames (B/P-frames). An unreliable protocol is clearly more like what these applications want.

What the applications do not necessarily want is to implement TCP-friendly congestion control. This is not only because congestion control can constrain performance, but also—perhaps more fundamentally—because implementing congestion control is very hard, as the long history of buggy TCP implementations makes clear [28, 29]. The applications might be willing to subject themselves to congestion control, not least for the good of the network, as long as it was easy to use and met their needs.

There are several ways this might be done. A library could implement congestion control over UDP, but this would be relatively slow, unportable, and might not provide access to features such as explicit congestion notification (ECN); such an approach would be all carrot and no stick. A second possibility would be to provide congestion control for unreliable applications at a layer below UDP, such as the Congestion Manager [3, 6]. Unfortunately, the Congestion Manager, as specified, can rely on application-level feedback about loss—not necessarily easy for applications to provide—and defines a single congestion control mechanism, whereas many of our target applications might prefer a choice between abruptly-changing AIMD algorithms and smoother rate-based algorithms such as TFRC [14].

These applications really need a new transport protocol: an unreliable datagram protocol with integrated congestion control. We set out to define this protocol. The goal was to provide a simple minimal congestion control protocol upon which other higher-level protocols could be built—UDP, plus the minimal TCP mechanisms necessary to support congestion control—and the result, the Datagram Congestion Control Protocol (DCCP) [12, 15, 21], has been approved for IETF standardization.

This paper describes DCCP, but that is not its main contribution. We expected it to be simple to provide an unreliable alternative to TCP. The issues turned out to be more

complex than we expected, and working through the design of an unreliable congestion-controlled protocol has given us a new appreciation for the way TCP's reliability, acknowledgement, and congestion control mechanisms intertwine into an apparently seamless whole. In particular, the loss of retransmissions, and their related feature cumulative acknowledgements, forced us to rethink almost every issue involving packet sequencing. Of course, TCP appears seamless only when you ignore its extensive evolution and extension, and we still believe that an unreliable protocol's simpler semantics forms a better base for layering functionality. We therefore discuss many of the issues we faced in designing a modern transport protocol—including some the TCP designers did not face as squarely, such as robustness against attack.

2 Application Requirements

Any protocol designed to serve a specific group of applications should consider what those applications are likely to need (although this needs to be balanced carefully against a desire to be future-proof and general). For the group of applications we are most concerned with, requirements include:

- **Choice of congestion control mechanism.** While our applications are usually able to adjust their transmission rate based on congestion feedback, they do have constraints on how this adaptation can be performed to minimize the effect on quality. Thus, they tend to need some control over the short-term dynamics of the congestion control algorithm, while being fair to other traffic on medium timescales. This control includes influence over which congestion control algorithm is used—for example, TFRC [14] rather than strict TCP-like congestion control. (TCP-Friendly Rate Control, or TFRC, is a congestion control mechanism that adjusts its sending rate more smoothly than TCP does, while maintaining long-term fair bandwidth sharing with TCP.)

Special application concerns include fairness at low packet sizes and at low rates, widely varying packet sizes, and stop-and-start communication.

First, many audio codecs send very small packets, down to 64 bits of payload; and in interactive audio, which has relatively tight latency constraints, combining samples into larger packets creates irritating, user-audible delay. Congestion control mechanisms should not unfairly penalize such flows for their small packet sizes, as long as the flows aren't sending small packets too frequently.

Second, video encoding standards often lead to application datagrams of widely varying size. MPEG key frames, or I-frames, are many times larger than incremental B- and P-frames. An encoder may generate packets at a fixed rate, but with orders-of-magnitude size variation.

Third, interactive applications are characterized by frequent stops and starts; for example, in telephony, one party generally shuts up while the other talks. Conventionally, congestion control reacts to application idleness by reducing the allowed send rate, since the application has ceased gathering feedback about the state of the network. However, applications generally want to return to their sustainable rate as soon as possible.

- **Low per-packet overhead.** Internet telephony and games in particular will tend to send small packets frequently, to achieve low delay and quick response time. Protocol overhead should not expand the packets unduly.

- **ECN support.** Explicit Congestion Notification [31] lets congested routers mark packets instead of dropping them. ECN capability must be turned on only on flows that react to congestion, but it is particularly desirable for applications with tight timing constraints, as there is often insufficient time to retransmit a dropped packet before its data is needed at the receiver.

- **Middlebox traversal.** UDP's lack of explicit connection setup and teardown presents unpleasant difficulties to network address translators and firewalls, with the result that some middleboxes don't let UDP through at all. Any new protocol should improve on UDP's friendliness to middleboxes.

2.1 Goals

Considering these application requirements, the evolution of modern transport, and our desire for protocol generality and minimality, we eventually arrived at the following feature goals for DCCP.

1. Modern congestion control. DCCP should conveniently support all the features of modern TCP congestion control implementations, including selective acknowledgements, explicit congestion notification (ECN), acknowledgement verification, and so forth.

2. Choice of congestion control mechanism. TCP congestion control quickly utilizes available bandwidth, but as a result its rate varies widely over time. This is a good tradeoff for file transfer, but not for streaming and interactive media; these applications, part of DCCP's target application set, might prefer to trade off peak utilization for a more slowly-changing rate. DCCP should allow applications to choose the congestion control mechanism to be used for a connection—for instance, TCP or TFRC [14]. Furthermore, DCCP should support experimentation with new congestion control mechanisms, from the low-speed TFRC variants discussed below to more radical changes such as XCP.

3. Self-sufficient congestion control. DCCP should provide applications with an API as simple as that of UDP. Thus, as in TCP, a DCCP implementation should be able to manage congestion control without application aid. This means that congestion control parameters must be negotiated in-band.

4. Tradeoffs between timing and reliability. Any API for sending DCCP packets will support some buffering, allowing the operating system to smooth out scheduling bumps. However, when the buffer overflows—the application's send rate is more than congestion control allows—a

smart application may want to decide exactly which packets should be sent. Some packets might be more valuable than others (audio data should usually be preferred to video, for example), or newer packets might be preferred to older ones: it depends on the application. DCCP should support not only naive applications, but also advanced applications that want fine-grained control over buffers and other tradeoffs between timing and reliability.

5. Accounting for non-congestion loss. DCCP is expected to be used on challenging links, including cellular and wireless technologies, where loss unrelated to congestion is common. Although there is no wide agreement on how non-congestion loss should affect send rates, or even how non-congestion loss can be reliably detected, DCCP should allow endpoints to declare that packets were lost for reasons unrelated to network congestion, such as receive buffer overflow.

6. Congestion control of acknowledgements. TCP doesn't enforce any congestion control on acknowledgements, except trivially via flow control. This is simultaneously too harsh and not harsh enough: high reverse-path congestion slows down the forward path, and medium reverse-path congestion may not even be detected, although it can be particularly important for bandwidth-asymmetric networks or packet radio subnetworks [7]. DCCP should thus detect and act on reverse-path congestion.

7. Minimal functionality. In line with the end-to-end argument and prior successful transport protocols in the TCP/IP suite, DCCP should not provide any functionality that could successfully be layered above it by the application (or an intermediate library). This yardstick helped determine what to leave out of the protocol; for instance, applications can easily layer multiple streams of data over a single unreliable connection. DCCP development focused on those features that support congestion control, and those features that cannot otherwise be layered on top.

8. No flow control. Taken to its logical conclusion, control over timing/reliability tradeoffs implies that DCCP should not require strict flow control. Receivers, like senders, may prefer to drop old data from their buffers in favor of new data as it arrives, or may prefer to implement an application-specific policy difficult to express at the transport level. In addition, flow control is nontrivial to get right: likely-mistaken flow control limits have been observed to lower peak transfer rates [1, 38]. DCCP should thus avoid imposing any flow control limitation separate from congestion control.

9. NAT transparency. DCCP must be transparent to network address translators, firewalls, and other middleboxes. Explicit connection setup and teardown is desirable, for example, since obvious protocol-designated initiation and completion sequences ease the implementation burden on firewalls and NATs. It also affected, for example, our mobility design, in which network addresses never appear in packet payloads.

10. Small headers. Packet headers in DCCP must be relatively compact even in the absence of header compression techniques. For example, spending 8 bytes to report an ECN Nonce is considered unacceptable overhead. Header overhead isn't critical for well-connected end hosts, but we want to support DCCP on ill-connected, low-powered devices such as cell phones.

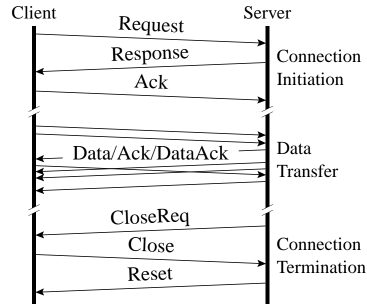


Figure 1: DCCP packet exchange overview.

11. Robustness and security. Network attackers are much more prevalent now than when the common TCP/IP transport protocols were originally designed, and for some time transport protocol evolution has been driven largely by security concerns. A modern protocol must be designed for robustness against attack. Robustness does not, however, require cryptographic guarantees; DCCP simply aims to provide a sequence number-based guarantee, similar to TCP. A DCCP connection is secure against third-party attacks like data injection and connection closure, *unless* the attacker can guess valid connection sequence numbers [26]. If initial sequence numbers are chosen sufficiently randomly [8], this means that attackers must snoop data packets to achieve any reasonable probability of success. However, we found a number of subtleties in applying sequence number security to an unreliable protocol; security conflicts directly with some of our other goals, including small headers, requiring a search for reasonable middle ground.

12. Robustness to denial-of-service. Finally, DCCP is designed to allow implementations to robustly resist many denial-of-service attacks.

3 DCCP Overview

DCCP, like TCP, is a unicast, connection-oriented protocol with bidirectional data flow. Connections start and end with three-way handshakes, as shown in Figure 1; datagrams begin with the 16-byte generic header shown in Figure 2. The Port fields resemble those in TCP and UDP. Data Offset measures the offset, in words, to the start of packet data; allocating 8 bits for this field means a DCCP header can contain more than 1000 bytes of option. The Type field gives the type of packet, and is somewhat analogous to parts of the TCP flags field. The names in Figure 1 correspond to packet types, of which DCCP specifies ten. Most packet types require additional information after the generic header, but before options begin; this design avoids cluttering the universal header with infrequently-used fields. There are no equiva-

lents to TCP's receive window and urgent pointer fields or its PUSH and URG flags, and TCP has no equivalent to CCVal (Section 6.4) or CsCov/Checksum Coverage (Section 6.2). Sequence and acknowledgement numbers are 48 bits long (but see Section 4.5).

4 SEQUENCE NUMBERS

We now turn to those properties of DCCP whose evolution surprised us, the most important being the entire interlocked set of issues surrounding sequence numbers. TCP's sequence numbers combine reliability, concision of acknowledgement, and bytestream semantics in a beautifully unified way; and as soon as we separated those properties—as we had to—the house of cards fell.

4.1 TCP sequence numbers

TCP uses 32-bit sequence numbers representing application data bytes. Each packet carries a sequence number, or seqno, and a cumulative acknowledgement number, or ackno.

The cumulative ackno indicates that all sequence numbers up to, but not including, the ackno itself have been received. The receiver guarantees that, absent a crash or application intervention, it will deliver that data to the application. Thus, the ackno succinctly summarizes the entire history of a connection. This succinctness comes at a price, however: the ackno provides no information about whether *later* data was received. Several interlocking algorithms, including fast retransmit, fast recovery, NewReno, and limited transmit [5], help avoid redundant retransmissions by inferring or tentatively assuming that data has been received. Such assumptions can be avoided if the sender is told exactly what data was received, and this more explicit approach is implemented by TCP selective acknowledgements, or SACK [10].

Sequence numbers generally correspond to individual bytes of application data, and variables measured in sequence numbers, such as receive and congestion windows, use units of data bytes. Thus, an endpoint may acknowledge *part* of a packet's contents (for instance, when a sender overflows the receiver's receive window), although this happens rarely in practice and may indicate an attempt to subvert congestion control [32]. Furthermore, TCP's congestion control algorithms generally operate on these byte-oriented variables in units of the *expected* packet size, which can lead to anomalies [2].

TCP connections contain other features that must be acknowledged, including connection setup and teardown, timestamps, ECN reports, and optional features like SACK. Connection setup and teardown is handled elegantly: SYN and FIN bits occupy sequence space, and are thus covered by the ackno. Each other feature, though, has its own acknowledgement mechanism. Each timestamp option contains an acknowledgement; ECN uses a TCP header bit (CWR) to acknowledge congestion reports; support for optional features is acknowledged via options, such as SACK-Permitted.

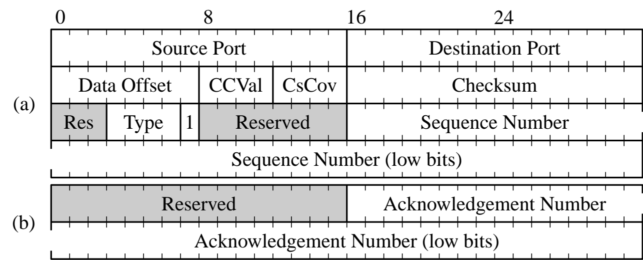


Figure 2: DCCP packet header. The generic header (a) begins every DCCP datagram. Individual packet types may add additional information, such as (b) an acknowledgement number. The packet header is followed by DCCP options, then payload; payload starts Data Offset words into the datagram.

Pure acknowledgements, which contain neither data nor SYN or FIN bits, do not occupy sequence space, and thus cannot be acknowledged conventionally. As a result, TCP cannot evaluate the loss rate for pure acknowledgements or detect or react to reverse-path congestion, except as far as high acknowledgement loss rates reduce the forward path's rate as well.

4.2 DCCP sequence numbers

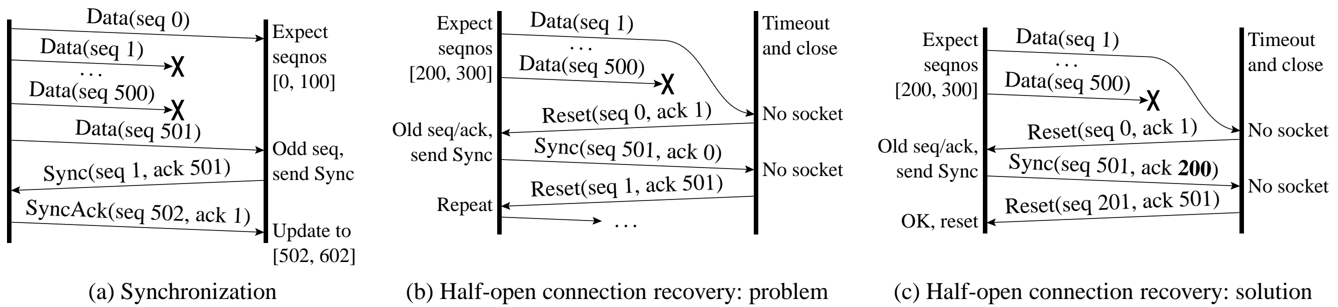
As a pure datagram protocol, UDP doesn't need sequence numbers; an application can layer them over the minimal UDP header, as in RTP. Congestion control algorithms must react to loss, however, and since DCCP cannot rely on application support (Goal 3), it must detect losses itself. This leads inevitably to sequence numbers. Every DCCP packet carries a sequence number in its header, and most packets additionally carry an acknowledgement number.

Cumulative acknowledgements don't make sense in an unreliable protocol, where data is never retransmitted as far as the transport layer is concerned. DCCP's ackno thus reports the *latest packet received*, not the earliest packet not received. This decision, which still seems inevitable, has tremendous consequences, since without a cumulative acknowledgement, there is no succinct summary of a connection's history. Additional connection options, including a run-length-encoded Ack Vector that says exactly which packets were received and a Loss Intervals option that reports intervals of non-dropped, non-marked packets, provide enough information to derive loss rates.

DCCP sequence numbers measure datagrams, not bytes, since DCCP applications send and receive datagrams rather than portions of a byte stream. This simplifies the expression of congestion control algorithms, which can work in units of packets—assuming that all packets are the same size, or that bottlenecks are in terms of packets, not bytes.

4.3 Synchronization

The obvious choice of assigning every packet a new sequence number causes a problem surprisingly difficult to solve: large bursts of loss can force endpoints out of sync. If a TCP connection is interrupted by network failure, its probe packets are retransmissions, and use expected sequence numbers. But in retransmissionless DCCP, each packet sent



(a) Synchronization (b) Half-open connection recovery: problem (c) Half-open connection recovery: solution

Figure 3: Recovering synchronization after bursts of loss.

during an outage uses a new sequence number. When connectivity is restored, each endpoint might have reached a sequence number wildly different from what the other expects.

We cannot eliminate expected sequence number windows, which are the line of defense protecting connections from attack (see Section 4.6). Instead, an explicit synchronization mechanism is introduced: an endpoint receiving an unexpected sequence or acknowledgement number sends a Sync packet asking its partner to validate that sequence number. The unexpected packets are not processed otherwise. (TCP in this situation would send a reset.) The other endpoint will process the Sync and reply with a SyncAck packet. When the original endpoint receives a SyncAck with a valid ackno, it updates its expected sequence number windows based on that SyncAck's seqno; see Figure 3(a) for an example. Because of their role in resynchronizing connections, Sync and SyncAck packets use more permissive expected sequence number windows than other packet types.

Some early versions of this mechanism synchronized using existing packet types, namely pure acknowledgements. However, in such a design, *mutually* unsynchronized endpoints can never resync, as there is no way to distinguish normal traffic from resynchronization attempts—both types of packet have either an unexpected seqno or an unexpected ackno. We considered using special options to get back into sync, but endpoints would have to partially parse options on possibly-invalid packets: a troublesome requirement, especially since earlier options in an options list might affect how later options are parsed. We considered preventing endpoints from sending data when they were at risk of getting out of sync, but this seemed fragile, imposed an artificial flow control limitation (contra Goal 8), and due to another choice, described in the next subsection, would not have helped. Explicit synchronization with unique packet types seems now like the only working solution.

The details are nevertheless subtle, and formal modeling revealed problems even late in the process. For example, consider the ackno on a Sync packet. In the normal case, this ackno should equal the seqno of the out-of-range packet, allowing the other endpoint to recognize the ackno as in its expected range. However, the situation is different when the out-of-range packet is a Reset, since after a Reset *the other endpoint is closed*. If a Reset had a bogus sequence number (due maybe to an old segment), and the resulting Sync echoed that bogus sequence number, then the endpoints would trade Syncs and Resets until the Reset's sequence number rose into the expected sequence number window (Figure 3(b)). Instead, a Sync sent in response to a Reset must set its ackno to the seqno of the latest valid packet received; this allows the closed endpoint to jump directly into the expected sequence number window (Figure 3(c)). As another example, an endpoint in the initial REQUEST state—after sending the connection-opening Request packet, but before receiving the Response—responds to Sync packets with Reset, not SyncAck. This helps clean up half-open connections, where one endpoint closes and reopens a connection without the other endpoint's realizing.

TCP senders' natural fallback to the known-synchronized cumulative ackno trivially avoids many of these problems. Some subtlety is still required to deal with half-open connections, but with fewer special cases.

4.4 Acknowledgements

DCCP's goals include applying congestion control to acknowledgements (Goal 6), negotiating congestion control features in band (Goal 3), and adding explicit connection setup and teardown (Goal 9). The former goal requires detecting acknowledgement loss; the second requires acknowledging each feature negotiation, which in TCP uses per-feature ad-hoc mechanisms. A single simple choice, motivated by TCP's inclusion of SYN and FIN in sequence space, seemed to solve all three problems at once: In DCCP, *every* packet, including pure acknowledgements, occupies sequence space, and uses a new sequence number.

This choice had several unintended consequences. For one, it forced us to deal with synchronization head-on, since *any* probe packets, not just data packets, might eventually cause endpoints to get out of sync. For another, an acknowledgement option such as Ack Vector combines, in one sequence space, information about *both* data packets and acknowledgements. Often this information should be separated: TCP does not consider acknowledgement loss when calculating fair rates, so neither should DCCP; and when calculating acknowledgement congestion control (Goal 6), DCCP shouldn't consider data packets.

But the most unusual property of DCCP acknowledgements is due to the more fundamental lack of cumulative

acknowledgements in an unreliable protocol. A TCP acknowledgement requires only a strictly bounded amount of state, namely the cumulative ackno. Although other SACK state may be stored, that state is naturally pruned by successful retransmissions. On the other hand, a DCCP acknowledgement contains potentially unbounded state. Ack Vector options can report every packet back to the beginning of the connection, bounded only by the maximum header space allocated for options. Since there are no retransmissions, the receiver—the endpoint reporting these acknowledgements—needs explicit help to prune this state. Thus, *pure acknowledgements must occasionally be acknowledged.* Specifically, the sender must occasionally acknowledge its receipt of an acknowledgement packet; at that point, the receiver can discard the corresponding acknowledgement information.

We seem to be entering an infinite regression—must acknowledgements of acknowledgements themselves be acknowledged? Luckily, no: an acknowledgement number indicating that a particular acknowledgement was received suffices to clean up state at the receiver; and this, being a single sequence number, uses bounded state at the sender.

Unreliability also affects the semantics of acknowledgement. In DCCP, an acknowledgement *never* guarantees that a packet's data will be delivered to the application. This is due to Goals 4 and 8, allowing a tradeoff between timeliness and reliability. Consider a streaming media receiver that prefers new data to old. If the receiver blocks for a while, it may find on resuming computation that more packets are locally enqueued than it can handle in the allotted time. DCCP allows the application, as part of the timeliness–reliability tradeoff, to drop the old data. For many reasons, though, this data must have been acknowledged already: acknowledgement options should, for congestion control purposes, report only losses and marks that happened in the network proper, and acknowledging packets only on application delivery would distort round-trip time measurements and unacceptably delay option processing. Thus, DCCP separates the concerns. The basic acknowledgement options, including Ack Vector, report header acknowledgement: a packet was received, processed, and found valid; its options were processed; and its data was enqueued for possible future delivery to the application. A separate run-length-encoded option, called Data Dropped, reports when basic acknowledgement differs from data acknowledgement: for example, when a packet was dropped in the receive buffer.

4.5 Sequence number length

How long should a sequence number be? Short sequence numbers lead to smaller headers (Goal 10), less bandwidth, and less endpoint state. On the other hand, short sequence numbers wrap more frequently—that is, long-lived connections must quickly reuse sequence numbers, leading potentially to old duplicate packets being accepted as new—and also make connections more vulnerable to attack. TCP's 32-bit per-byte sequence numbers already have wrapping problems at gigabit network speeds (a problem addressed by the timestamp option).

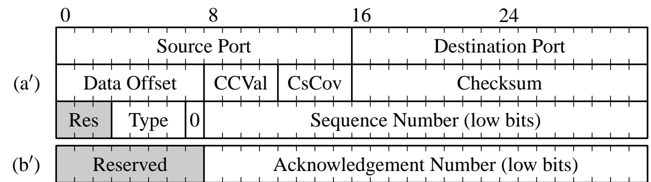


Figure 4: DCCP header with short sequence numbers. See also Fig. 2.

DCCP originally chose to use short 24-bit sequence numbers. We reasoned that fast connections would favor fewer large packets over many small packets, leaving packet rates low. This was, of course, a mistake. A datagram protocol cannot force its users to use large packet sizes, and padding packets with garbage data simply to avoid sequence number wrapping would be perverse; but absent packet length restrictions, 24 bits are too few: a 10 Gb/s flow of 1500-byte packets will send 2^{24} packets in just 20 seconds.

We considered several solutions. The header could be rearranged, albeit painfully, to allow 32-bit sequence numbers, but this doesn't provide enough cushion to avoid the issue. TCP's timestamp option is a bad model—verbose, complex, and still vulnerable to attack. Even a more concise and consistent timestamp would force implementations to parse the options area before determining whether the packet had a valid sequence number.

The simplest and best solution was simply to lengthen sequence numbers, specifically to 48 bits (64 would have crowded out other header fields). A connection using 1500-byte packets would have to send more than 14 petabits a second before wrapping 48-bit sequence numbers unsafely fast (that is, more than once every 2 minutes).

However, *forcing* the resulting overhead on all packets was considered unacceptable; consider audio codecs, in which 8-byte payloads are not untypical. Endpoints must be able to choose between short and long sequence numbers.

At first, connection initiation would negotiate whether sequence numbers would be long or short, but this required that the client endpoint know ahead of time how fast a connection would go. The solution, once found, was cleaner: DCCP sequence numbers are 48 bits long, but some packets may leave off the upper 24 bits of the sequence number (Figure 4). The receiver extends a 24-bit fragment into a full sequence number by comparing it with an expected sequence number. The following procedure takes a 24-bit fragment s and an expected sequence number r, and returns s's 48-bit extension. It includes two types of comparison: absolute, written $<$, and circular mod 2^{24}, written $\lessdot$.

$r_{\text{low}} := r \bmod 2^{24}$, $r_{\text{high}} := \lfloor r/2^{24} \rfloor$;
if ($r_{\text{low}} \lessdot s < r_{\text{low}}$) // *s incremented past* $2^{24}-1$
    return $((r_{\text{high}}+1) \bmod 2^{24}) \times 2^{24} + s$;
else if ($s \lessdot r_{\text{low}} < s$) // *s decremented past* 0 *(reordering)*
    return $((r_{\text{high}}+2^{24}-1) \bmod 2^{24}) \times 2^{24} + s$;

else
    return $r_{\text{high}} \times 2^{24} + s$;

Connection initiation, synchronization, and teardown packets always use 48-bit sequence numbers. This ensures that the endpoints agree on sequence numbers' full values, and greatly reduces the probability of success for some serious attacks; see below. But data and acknowledgement packets—exactly those packets that will make up the bulk of the connection—may, if the endpoint approves, use 24-bit sequence numbers instead, trading maximum speed and incremental attack robustness for lower overhead. Although a single sequence number length would be cleaner, we feel the short sequence number mechanism is one of DCCP's more successful features. Good control over overhead is provided at moderate complexity cost, without opening the protocol unduly to attack.

Another protocol mechanism that reduces overhead is the optional acknowledgement number, which potentially reduces overhead for unidirectional flows.

4.6 Robustness to attack

Robustness to attack is now a primary protocol design goal. Attackers should find it no easier to violate a new protocol's connection integrity—by closing a connection, injecting data, moving a connection to another address, and so forth—than to violate TCP's connection integrity. Unfortunately, this is not a high bar.

TCP guarantees *sequence number security*. Successful connection attacks require that the attacker know (1) each endpoint's address and port and (2) valid sequence numbers for each endpoint. Assuming initial sequence numbers are chosen well (that is, randomly) [8], reliably guessing sequence numbers requires snooping on traffic. Snooping also suffices: any eavesdropper can easily attack a TCP connection [26]. Applications desiring protection against snooping attacks must use some form of cryptography, as in IPsec, *ssh*-style end-to-end encryption, or TCP's MD5 option.

Of course, a non-snooping attacker can always try their luck at guessing sequence numbers. If an attacker sends N attack packets distributed evenly over a space of 2^L sequence numbers (the best strategy), then the probability that one of these attack packets will hit a window W sequence numbers wide is $WN/2^L$; if the attacker must guess both a sequence number and an acknowledgement number, with validity windows W_1 and W_2, the success probability is $W_1 W_2 N/2^{2L}$. In TCP, data injection attacks require guessing both sequence and acknowledgement numbers, but connection reset attacks are easier—a SYN packet will cause connection reset if its sequence number falls within the relevant window. (A similar attack with RST packets was recently publicized, but this is somewhat easier to defend against.) Recent measurements report a median advertised window of approximately 32 kB [25]; with $W = 32768$, this attack will succeed with more than 50% probability when $N = 65536$. This isn't very high, and as networks grow faster, receive window widths are keeping pace, leading to easier and easier attacks.

DCCP's 48-bit sequence numbers and support for explicit synchronization make reset attacks much harder to execute. For example, DCCP is immune to TCP's SYN attack; if a Request packet hits the sequence window of an active connection, the receiving endpoint simply responds with a Sync. The easiest reset-like attack is to send a Sync packet with random sequence and acknowledgement numbers. If the ackno by chance hits the relevant window, the receiver will update its other window to the attacker's random sequence number. In many cases another round of synchronization with the true endpoint will restore connectivity, but lucky attacks will lead to long-term loss of connectivity, since the attacked endpoint will think all of its true partner's packets are old. But even given a large window of $W = 2000$ (nearly 3 MB worth of 1500-byte packets), an attacker must send more than 10^{11} packets to get 50% chance of success.

Unfortunately, the goal of reducing overhead conflicts with security. DCCP Data packets may use 24-bit sequence numbers, and contain no acknowledgement number. As a result, it is quite easy to inject data into a connection that allows 24-bit sequence numbers: given the default window of $W = 100$, an attacker must send $N \approx 83000$ Data packets to get 50% chance of success. An application can reduce this risk simply by not asking for short sequence numbers, and data injection attacks seem less dangerous than connection reset attacks; the attacker doesn't know where in the stream their data will appear, and DCCP applications must already deal with loss (and, potentially, corruption).

But unless we are careful, data injection might cause connection reset. For example, an injected Data packet might include an invalid option whose processing would reset the connection. Several aspects of the protocol were modified to prevent this kind of escalation; at this point, no Data packet, no matter how malformed its header or options, should cause a DCCP implementation to reset the connection, or to perform transport-level operations that might eventually lead to resetting the connection. For instance, many options must be ignored when found on a Data packet. In retrospect, these modifications accord with the TCP Robustness Principle, "be conservative in what you send, and liberal in what you accept". Although careful validity checking with harsh consequences for deviations may seem appropriate for a hostile network environment, attackers can exploit that checking to cause denial-of-service attacks. It is better to keep to the principle and ignore any deviations that attackers might cause.

4.7 Discussion

While it seems possibly convenient to base application-level sequence numbers off DCCP's packet sequence numbers, the combination of acknowledgements and data packets in a single space makes it difficult. An NDP Count option allows senders to calculate how many *data* packets have been

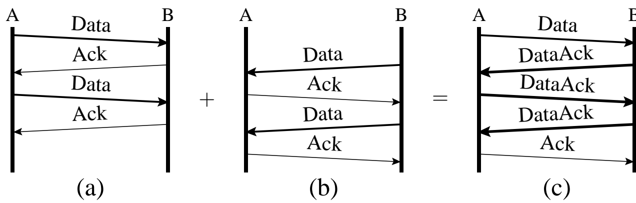


Figure 5: (a) An A-to-B half-connection and (b) a B-to-A half-connection combine into (c) a full connection with piggybacked data and acknowledgements.

sent, as opposed to non-data packets, even in the presence of loss. An analysis of RTP indicated, however, that applications' sequence number concepts are just different enough to make direct correspondence nonuseful; applications will just have to layer their own sequence numbers on top.

Not all comparisons between TCP's cumulative acknowledgements and bytestream-oriented sequence numbers and DCCP-style unreliable, packet-oriented sequence numbers come out in favor of TCP. For example, bytestream-oriented sequence numbers make it ambiguous whether an acknowledgement refers to a packet or its retransmission, which has led to a cottage industry in acknowledgement disambiguation and recovery from spurious retransmissions [24].

5 Connection Management

This section describes DCCP properties, including several with interesting differences from TCP, that do not directly concern sequence numbers.

5.1 Asymmetric communication

DCCP, like TCP, provides a single bidirectional connection: data and acknowledgements flow in both directions. However, many DCCP applications will have fundamentally asymmetric data flow—for example, in streaming media almost all data flows from server to client—and DCCP should be able to optimize for that case. For example, in a bidirectional connection, each endpoint might keep detailed SACK-like acknowledgement information about its partner's packets. When data flows unidirectionally from A to B, however, only B's records need maintain that level of detail: maintaining a run-length-encoded vector that covers only acks would waste bandwidth, state, and processing time.

DCCP models asymmetric data flow by dividing data transfer into two logical *half-connections*. Each half-connection consists of data packets from one endpoint, plus the corresponding acknowledgements from the other. When communication is bidirectional, both half-connections are active, and acknowledgements can often be piggybacked on data packets; see Figure 5. Acknowledgement format is determined by the governing half-connection, which might for example require detailed Ack Vector information. But a half-connection that has sent no data packets for some time (0.2 seconds or 2 RTTs, whichever is greater), and that has no outstanding acknowledgements, is said to be *quiescent*. There is no need to send acknowledgements on a quiescent half-connection. When the B-to-A half-connection goes quiescent (B stops sending data), A can also stop acknowledging B's packets, except as necessary to prune B's acknowledgement state (Section 4.4). TCP, of course, has it easier, and devolves naturally into unidirectional communication. TCP acknowledgements occupy no sequence space, so it is neither useful nor possible to acknowledge them; and since data retransmissions clean up old ack state, a "quiescent TCP half-connection"—in which all data has been acknowledged—occupies minimal state on both endpoints by definition.

Half-connections turn out to be a useful abstraction for managing connection state. DCCP runs with this idea: each half-connection has an independent set of variables and features, including congestion control mechanism. Thus, a single DCCP connection might consist of one half-connection using TCP-like congestion control and one using TFRC; or, more realistically, of two TFRC half-connections with different parameters. It makes sense conceptually and in the implementation to group information related to a data stream with information about its reverse path.

5.2 Feature negotiation

A DCCP connection's endpoints must agree on a set of parameters, most clearly the congestion control mechanisms to be used. Both endpoints have capabilities (the mechanisms they implement) and application requirements (the mechanisms the application would prefer), both of which should both be able to influence the outcome. Furthermore, the application cannot be relied upon to negotiate agreement, so negotiation must take place in band (Goal 3).

DCCP thus provides options for negotiating the values of general *features*. A feature is simply a per-endpoint property on whose value the endpoints must agree. Examples include each half-connection's congestion control mechanism and whether short sequence numbers are allowed.

Feature negotiation involves two option types: Change options open feature negotiation; Confirm options, which are sent in response, name the new values. Change options are retransmitted as necessary for reliability. Each feature negotiation takes place in a single option exchange (our initial design involved multiple back-and-forth rounds, but this proved fragile). This isn't overly constraining, since complex preferences can be described in the options themselves: Change and Confirm options can contain preference lists, which the endpoints analyze to find a best match.

With hindsight, the decision to provide generic reliable feature negotiation has allowed additional functionality to be added easily without the need to consider interactions between feature negotiation, congestion control, and the differing acknowledgement styles required by each congestion control mechanism.

5.3 Mobility and multihoming

Mobility and multihoming, with their integration between network and transport layers, are some of the very few features that cannot be layered on top of an unreliable protocol. Mobility, at least, can be implemented entirely at the network layer, but choosing the transport layer has advantages [34]: the transport layer is naturally aware of address shifting, so its congestion control mechanism can respond appropriately; also, transport-layer mobility avoids triangle routing issues. We were thus directed to develop a mobility and multihoming mechanism for DCCP. The mechanism had to prevent "connection hijacking", where an attacker moves one endpoint of a victim connection to its own IP address, *even when the attacker can passively snoop the connection*. This is a departure from DCCP's basic security model; we reason that hijacking is fundamentally more serious than data injection or connection reset. (Of course, an in-path attacker, such as a compromised router, that could inject and alter traffic could already hijack the connection, mobility or no.) There is no way to prevent a passively-snooping attacker from hijacking without using some form of cryptography.

Happily, mobility and multihoming are one of the few cases where unreliability makes a problem easier. Reliable transport must maintain in-order delivery even across multiple addresses. As a consequence, changing a connection's address set requires tight integration with the transport layer [34]. Unreliable transport, however, doesn't guarantee in-order delivery, or any delivery at all, and coordination can therefore be quite loose. DCCP's mobility and multihoming mechanism simply joins a set of component connections—all of which may have different endpoint addresses, ports, sequence numbers, and even connection features—into a single *session* entity. To add a new address, an endpoint opens a new DCCP connection, including on its Request an option for attaching to an existing session. Such session options are protected against forgery and replay by nonces and cryptographic signatures. Most DCCP code can treat component connections as independent; for instance, each connection has its own congestion control state. The only code that differs involves the socket layer, where transport interacts with application, and the difference is slight: whereas most transport state applies to a component connection, the socket corresponds to a session. Thus, data sent to the socket can be distributed arbitrarily among component connections, and data received from any component connection is enqueued on the shared socket. This design resembles previous work on session-layer mobility management [22, 35], but thanks to unreliability, can add multihoming support while simplifying the basic abstractions.

5.4 Extensions and subsets

To facilitate future extension, protocol implementations generally ignore unknown options, unknown packet types, the values of reserved header fields, and so forth, and DCCP's feature mechanism helps further by providing a general means to acknowledge an extension's presence. But some extensions must effectively be mandatory. For example, consider DCCP subsets: profiles, designed for use by embedded systems implementors, that curtail option choice, option ordering, and feature values, essentially reducing protocol complexity. An embedded system implementing a subset cannot communicate with an endpoint that does not understand that subset. Thus, DCCP's Mandatory option, which indicates that the immediately following option is mandatory: if the receiving endpoint cannot fully process that option, it will reset the connection. (Compare SCTP's chunk types [30].) Mandatory is a prime example of an option that must be ignored on Data packets (Section 4.6).

5.5 Denial-of-service attacks

In a transport-level denial-of-service attack, an attacker tries to break an endpoint's network stack by overwhelming it with data or calculations. For example, an attacker might send thousands of TCP SYN packets from fake (or real) addresses, filling up the victim's memory with useless half-open connections. Generally these attacks are executed against servers rather than clients. DCCP addresses potential denial-of-service attacks by pushing state to the clients when possible, and by allowing endpoints to rate-limit responses to invalid packets. For example, when responding to a Request packet, a server can encapsulate all of its connection state into an "Init Cookie" option, which the client must echo when it completes the three-way handshake. Like TCP's SYN cookies [9], this lets the server keep no information whatsoever about half-open connections; unlike SYN cookies, it can encapsulate lots of state. DCCP servers can often shift Time-Wait state onto clients, due to restrictions on how connection termination exchanges may begin. (Time-Wait state remains at an endpoint for at least two minutes to prevent confusion in case the network delivers packets late.) The protocol allows rate limits whenever an attacker might force an endpoint to do work; for example, there are optional rate limits on the generation of Reset and Sync packets. Finally, as described above, DCCP has been engineered more generally to prevent non-snooping attackers from resetting existing connections.

5.6 Formal modeling

The initial DCCP design was completed without benefit of formal modeling. As our work progressed, however, we made use of a semi-formal exhaustive state search tool and two formal tools, a labeled transition system (LTSA) model and an independently-developed colored Petri net (CPN) model from the University of South Australia [37]. These tools, and particularly the colored Petri net model, were extremely useful, revealing several subtle problems in the protocol as we had initially specified it.

The most important semi-formal tool was, unsurprisingly, shifting from reasoning via state diagrams to detailed pseudocode that defined how packets should be processed. The

resulting precision showed us several places where our design could lead to deadlock, livelock, or other confusion. An ad hoc exhaustive state space exploration tool was then developed to verify that the pseudocode worked as expected; examining its output led to further refinements, especially to the mechanism for recovering from half-open connections. The LTSA model—which included states, packets, timers, and a network with loss and duplication, but not sequence numbers—was used to more formally examine the specification for progress and deadlock freedom. It found a deadlock in connection initiation, which we fixed. The CPN model went into more depth, in particular by including sequence numbers, with impressive results. This model found the half-open connection recovery problem described in Figure 3(b), a similar problem with connections in timewait state, and a problem with the short-sequence-number extension code in Section 4.5 (we initially forgot reordering). These problems involved chatter, rather than deadlock: a connection would eventually recover, but only after sending many messages and causing the verification tool's generalized state space to explode in size. Thus, as the protocol improved the verifier ran more quickly!

Our experience with protocol modeling was quite positive, especially in combination with the clear explanation of pseudocode, and next time, we would seek out modeling experts earlier in the design process.

6 Congestion Control

With DCCP, the application has a choice of congestion control mechanisms. Many unreliable applications might prefer TFRC congestion control, avoiding TCP's abrupt halving of the sending rate in response to congestion, while other applications might prefer a more aggressive TCP-like probing for available bandwidth.

This selection is done by using Congestion Control IDs (CCIDs) to indicate the choice of standardized congestion control mechanisms, with the connection's CCID being negotiated at connection start-up. This profile-based selection allows the introduction of CCID-specific options and features, which avoid polluting the global option and feature space. For example, option numbers 128 to 255 have CCID-specific meaning; this space is further split between the two half-connections that might be relevant for a piggybacked data-plus-ack.

6.1 Misbehaving receivers

Internet congestion control is voluntary in the sense that few, if any, routers actually enforce congestion control compliance. Unfortunately, some endpoints, particularly receivers, have incentives to violate congestion control if that will get them their data faster. For example, misbehaving receivers might pretend that lost packets were received or that ECN-marked packets were received unmarked, or even acknowledge data before it arrives [32]. TCP's semantics deter many of these attacks, since missing data violates the expectation of reliability and must therefore be handled by the application. However, DCCP applications must tolerate loss in any case, making deliberate receiver misbehavior more likely.

The protocol must therefore be designed to allow the detection of deliberate misbehavior. In particular, senders must be able to verify that every acknowledged packet was received unmarked. To do this the sender provides an unpredictable nonce with each packet; the receiver echoes an accumulation of all relevant nonces in each acknowledgement [32].

DCCP, like TCP, uses the ECN Nonce for this purpose, which encodes one bit of unpredictable information that is destroyed by loss or ECN marking [36]. All acknowledgement options contain a one-bit nonce echo, set to the exclusive-or of the nonces of those packets acknowledged as received non-marked. However, unlike in TCP, calculating and verifying this nonce echo presents no difficulties. The TCP nonce echo applies to the cumulative ack, and thus covers every packet sent in the connection. In the presence of retransmission and partial retransmission, however, a sender can never be sure exactly which packets were received, as retransmissions have the same sequence numbers as their originals. Thus, the TCP nonce echo and verification protocol must specially resynchronize after losses and marks. None of this is necessary in DCCP, where there are no retransmissions—every packet has its own sequence number—and no cumulative ack: options such as Ack Vector explicitly declare the exact packets to which they refer.

An endpoint that detects egregious misbehavior on its partner's part should generally slow down its send rate in response. An "Aggression Penalty" connection reset is also provided, but we recommend against its use except for apocalyptic misbehavior; after all, an attacker can confuse an endpoint's nonce echo through data injection attacks.

Several other DCCP features present opportunities for receiver misbehavior. For example, Timestamp and Elapsed Time options let a receiver declare how long it held a packet before acknowledging it, thus separating network round-trip time from end host delay. The sender can't fully verify this interval, and the receiver has reason to inflate it, since shorter round-trip times lead to higher transfer rates. Thus far we have addressed such issues in an ad hoc manner.

6.2 Partial checksums and non-congestion loss

Several of our target applications, particularly audio and video, not only tolerate corrupted data, but prefer corruption to loss: passing corrupt data to the application may improve its performance as far as the user is concerned [18, 33]. While some link layers essentially never deliver corrupt data, others, such as cellular technologies GSM, GPRS, and CDMA2000, often do; furthermore, link-layer mechanisms for coping with corruption, such as retransmission (ARQ), can introduce delay and rate variability that applications want even less than corruption [11]. DCCP therefore follows the UDP-Lite protocol [23] in allowing its checksum

to cover less than an entire datagram. Specifically, its checksum coverage (CsCov) field allows the sender to restrict the checksum to cover just the DCCP header, or both the DCCP header and some number of bytes from the payload. A restricted checksum coverage indicates to underlying link layers that some corruption is acceptable, and corrupt datagrams should be passed on rather than dropped or retransmitted (as long as the corruption takes place in the unprotected area).

The motivation for partial checksums follows that of UDP-Lite, but is perhaps more compelling in DCCP because of congestion control. Wireless link technologies often exhibit an underlying level of corruption uncorrelated with congestion, but endpoints treat all loss as indicative of congestion. Various mechanisms have been proposed for differentiating types of loss, or for using local retransmissions to compensate [4]. While it isn't clear in the end how one *should* differentiate congestion responses for these types of loss, we believe that protocols should at least allow the differentiation.

Thus, DCCP allows receivers to report corruption separately from congestion, when the corruption is restricted to packet payload. (Payload corruption may be detected with a separate CRC-based Payload Checksum option; all packets with corrupt headers must be dropped and reported as lost.) This uses the same mechanism as other types of non-network-congestion loss, such as receive buffer drops: the packet is reported as received, and its ECN Nonce is included in the relevant acknowledgement option's nonce echo, but a separate Data Dropped option reports the corruption. Congestion control mechanisms currently treat corruption as they would treat ECN marking, thus initiating a congestion response, but at least the information is available for future use.

6.3 CCID 2: TCP-like Congestion Control

DCCP's CCID 2 provides a TCP-like congestion control mechanism, with its corresponding abrupt rate changes and ability to take advantage of rapid fluctuations in available bandwidth. CCID 2 acknowledgements use the run-length-encoded Ack Vector option, which is essentially a version of TCP's SACK. Its congestion control algorithms likewise follow those of SACK TCP, and maintain similar variables: a congestion window "cwnd", a slow-start threshold, and an estimate of the number of data packets outstanding [10].

Unlike SACK TCP, however, CCID 2 can detect and respond to reverse-path congestion. CCID 2 maintains a feature called the Ack Ratio, which equals a rough ratio of data packets per acknowledgement. TCP-like delayed-ack behavior is provided by the default Ack Ratio of 2. As a CCID 2 sender detects lost and marked acknowledgements, however, it manipulates the Ack Ratio so as to reduce the acknowledgement rate in a very roughly TCP-friendly way. Ack Ratio always meets three constraints: (1) it is an integer; (2) it does not exceed $\text{cwnd}/2$, rounded up, except that Ack Ratio 2 is always acceptable; and (3) it is two or more for a congestion window of four or more packets. The sender changes Ack Ratio within those constraints as follows: for each congestion window of data with lost or marked acks, Ack Ratio is doubled; and for each $\text{cwnd}/(R^2 - R)$ consecutive congestion windows of data with no lost or marked acks (where R is the current Ack Ratio), it is decreased by 1. (This second formula is derived as follows: We want to increase the number of acks per congestion window, namely cwnd/R, by one per congestion-free window. However, since R is an integer, we instead find a k so that, after k congestion-free windows, $\text{cwnd}/R + k = \text{cwnd}/(R-1)$.)

Unfortunately, DCCP's shared sequence number space does not allow a sender to distinguish whether a missing packet was a data packet or an acknowledgement. CCID 2 senders thus assume that every missing receiver packet was an acknowledgement, absent NDP Count or other options that can provide better information. In most cases, this assumption is safe; for example, if the connection is unidirectional, all of the receiver's packets are in fact acknowledgements. One bad case is a bidirectional connection where the CCID 2 receiver is sending about as many data packets as acks, so that the CCID 2 sender mischaracterizes many observed losses that were actually data packets. The Ack Ratio will be set to about 0.62 times its true value [12], leading to a mild increase in burstiness for the CCID 2 sender. To solve this issue the CCID 2 sender could simply rate-pace its packets, as it probably should anyway.

6.4 CCID 3: TFRC Congestion Control

TFRC congestion control in DCCP's CCID 3 uses a completely different approach. Instead of a congestion window, a CCID 3 sender uses a sending rate, and the receiver sends feedback to the sender roughly once per round-trip time reporting the loss event rate calculated by the receiver. The sender uses the reported loss event rate to determine its sending rate. If the sender receives no feedback from the receiver for several round-trip times, then the sender halves its sending rate.

This is reasonably straightforward, and does not require the reliable delivery of feedback packets, as long as the sender trusts the receiver's reports of the loss event rate. Since acknowledgements are so limited—to one per round-trip time—there is no need for acknowledgement congestion control. However, a mere loss rate is ripe for abuse by misbehaving receivers. Thus, CCID 3 requires instead that the receiver report a set of *loss intervals*, the quantities from which TFRC calculates a loss rate. Each loss interval contains a long tail of non-dropped, non-marked packets; the Loss Intervals option reports each such tail's nonce echo, thus allowing the sender to verify the acknowledgement.

Note that the feedback information required by TFRC is substantially different from that required by TCP-style congestion control. A protocol whose basic feedback mechanism is not sufficiently flexible could have difficulties in the

future as the state of the art of congestion control evolves.

TFRC also requires that data senders attach to each data packet a coarse-grained "timestamp" that increments every quarter-round-trip time. This timestamp allows the receiver to group losses and marks that occurred during the same round-trip time into a single congestion event. Such a timestamp could obviously be included as an option, but at the cost of 4 bytes per packet. Instead, CCID 3 attaches the timestamp to a 4-bit protocol header field, CCVal, reserved for use by the sender's congestion control mechanism. Such a small field requires care to avoid wrapping problems; we considered this worth it to avoid the overhead.

6.5 TFRC's small-packet variant

TCP offers a simple semantics to applications of congestion-controlled, reliable byte-stream transfer. In contrast to the file transfer, email and web applications that use TCP, applications that use unreliable transfer generally forgo reliable delivery because they are constrained by a playback time (e.g., some forms of voice, video, and on-line games), and this rules out some of the long delays that are possible in retransmitting dropped packets.

As was discussed above, the congestion control mechanisms for unreliable applications can be affected by application-related tradeoffs such as a desire to avoid abrupt decreases in the allowed sending rate, versus a desire to make use of available bandwidth as promptly as possible.

Another application-related issue that affects the development of congestion control mechanisms for unreliable transfer is that some applications (e.g., some forms of voice) wish to send frequent small packets. Designing congestion control mechanisms for such applications involves a fundamental question about whether the congested points in the network are limited by their sending rate in packets per second (e.g., CPU cycles at routers), or by their sending rate in bytes per second (e.g., bandwidth). The development of a small-packet variant of TFRC is based on the assuption that the limitation today is generally one of bandwidth rather than of CPU cycles, and that as a reault small-packet applications should be able to receive the same bandwidth in bytes per second as large-packet TCP connections would experiencing the same packet drop rate.

In TCP, for a given packet drop rate, the allowed sending rate can be expressed in terms pf packets per round-trip time; a TCP connection that uses larger packets is allowed a proportionally larger sending rate in bytes per second. However, because TCP offers a byte-stream semantics to applications, TCP is responsible for assembling data into packets, and TCP generally assembles packets to be as large as possible; the effect of the packet size on the sending rate has not been a pressing issue in the development of TCP congestion control.

For congestion control for unreliable transfer, however, it is a different issue, with a wide range of packet sizes, and with packetization happening at the application rather than in the transport protocol. And for the flows that want to send frequent small packets (e.g., the frequent 14-byte data packets that can be sent by some voice applications), the effect of the packet size on the allowed sending rate can be a critical issue [13, 19].

However, even given an agreement that small-packet and large-packet flows that experience the same congestion should receive the same bandwidth in bytes per second, there is the complicating factor of the lack of standardization of router behavior in the Internet, in terms of congestion control. Some routers in the Internet might have Drop-Tail queues in units of packets, or Active Qeueue Management mechanisms in packet mode, where small packets and large packets are equally likely to be dropped. Other routers in the Internet might have Drop-Tail queues in units of bytes, or Active Qeueue Management mechanisms in byte mode, where small packets are less likely to be dropped than large packets. As we see below, this complicates considerably the job of designing congestion control mechanisms for small-packet flows. In particular, a flow can't necessarily infer, from its own congestion indications, anything about the congestion indications seen by competing traffic.

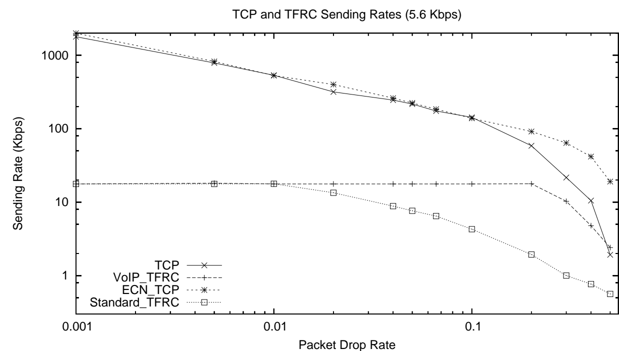


Figure 6: Send Rate vs. Packet Drop Rate: 14B TFRC Segments (5.6 Kbps Maximum TFRC Data Sending Rate)

The figure above shows simulations with both TCP and TFRC connections. The x-axis represents the steady-state packet drop rate p in each simulation. The TCP flows use 1460-byte data packets, while the TFRC flows use 14-byte data packets. For this environment, the flow using standard TFRC receives considerably less bandwidth that the TCP flows. A fourth line shows a TFRC flow using a Small-Packet variant of TFRC (also called VoIP TFRC), designed to allow small-packet TFRC flows to receive the same sending rate in bytes per second as large-packet TCP or TFRC flows sharing the same packet drop rate. As the figure shows, in this scenario the small-packet flow receives a very low throughput with standard TFRC, but does well with the Small-Packet variant of TFRC, as designed.

In contrast, the figure above shows TCP and TFRC connections when each simulation has a fixed *byte* drop rate instead of a fixed *packet* drop rate. Again, the TFRC flows use 14-byte data packets. In each simulation, the TCP flow uses the best packet size for that byte drop rate - in this scenario,

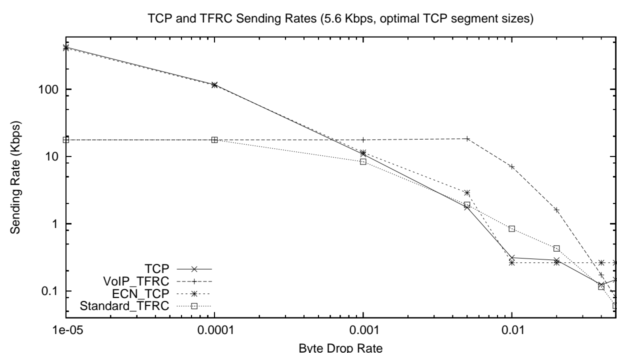


Figure 7: Sending Rate vs. Byte Drop Rate

TCP will sometimes have a higher sending rate, and receive higher throughput, when it uses a smaller packet size.

The figure shows that for this scenario of a fixed byte drop rate, standard TFRC gives roughly the desired performance of fairness between the small-packet TFRC flow and the possibly-large-packet TCP flow, while the Small-Packet variant of TFRC allows the small-packet TFRC flow to receive much more than its 'share' of the bandwidth. Similar results can be obtained from simulations with Drop-Tail queues in units of bytes, or with queues using Active Queue Management in byte mode.

All of this leaves open the question of appropriate congestion control mechanisms for small-packet flows in the Internet.

6.6 Future congestion control mechanisms

Another issue raised by transport protocols for unreliable transfer are about the needs of the applications. The applications (e.g., video, audio, on-line games) might want to have sharp changes in the sending rate from one RTT to the next (changes in the video contents); or might want to have faster startup and fast sending after an idle period, etc. There are fundamental issues we don't yet understand, about how far congestion control mechanisms for best-effort traffic can be pushed to deal with these application-level issues; and what the consequences might be for aggregate traffic if congestion control mechanisms are pushed too far.

7 Related Work

We are not able to cite here the related work on all of the issues touched on in this paper.

Related work on the architectural and technical issues in the development of new transport protocols includes papers on RTP [17], RTSP [16], SCTP [30], and UDP-Lite [23]. There is also a body of research on the development of new congestion control mechanisms for high-bandwidth environments, or with more explicit feedback from routers, but that is not directly relevant here.

8 Conclusions

It might reasonably be assumed that designing an unreliable alternative to TCP would be a rather simple process; indeed we made this assumption ourselves. However, TCP's congestion control is so tightly coupled to its reliable semantics that few TCP mechanisms are directly applicable without substantial change. For example, the cumulative acknowledgement in TCP serves many purposes, including reliability, liveness, flow control and congestion control. There does not appear to be a simple equivalent mechanism for an unreliable protocol.

The current Internet is a hostile environment, and great care needs to be taken to design a protocol that is robust. TCP has gained robustness over time, and it is important to learn from its mistakes. However, the problem for an unreliable protocol is actually harder in many ways; the application semantics are not so well constrained, and there seem to be more degrees of freedom for an attacker.

The current Internet is also a confused environment; unless a protocol wishes to condemn itself to irrelevance, its design must make it easy to deploy in a world of NATs, firewalls and justifiably paranoid network administrators.

In this paper we have attempted to sketch out many of the issues that arose in the design of DCCP, some of them obvious, others more subtle. We have deliberately avoided describing all the details of DCCP; the interested reader is referred to the DCCP specification.

Acknowledgements

DCCP has benefitted from conversations and feedback with many people not cited here to maintain the fig leaf of anonymity.

References

[1] M. Allman. A Web server's view of the transport layer. *ACM Computer Communication Review*, 30(5), Oct. 2000.

[2] M. Allman. TCP byte counting refinements. *ACM Computer Communication Review*, 29(3), July 1999.

[3] H. Balakrishnan and S. Seshan. The Congestion Manager. RFC 3124, Internet Engineering Task Force, June 2001.

[4] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz. A comparison of mechanisms for improving TCP performance over wireless links. *IEEE/ACM Trans. on Networking*, 5(6):756–769, Dec. 1997.

[5] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R. H. Katz. TCP behavior of a busy Internet server: Analysis and improvements. In *Proc. IEEE INFOCOM 1998*, pages 252–262, Mar. 1998.

[6] H. Balakrishnan, H. Rahul, and S. Seshan. An integrated congestion management architecture for internet hosts. In *Proc. SIGCOMM 1999*, Aug. 1999.

[7] H. Balakrishnan, V. N. Padmanabhan, G. Fairhurst, and M. Sooriyabandara. TCP performance implications of network path asymmetry. RFC 3449, Internet Engineering Task Force, Dec. 2002.

[8] S. Bellovin. Defending against sequence number attacks. RFC 1948, Internet Engineering Task Force, May 1996.

[9] D. J. Bernstein. SYN cookies. Web page. `http://cr.yp.to/syncookies.html`.

[10] E. Blanton, M. Allman, K. Fall, and L. Wang. A conservative selective acknowledgment (SACK)-based loss recovery algorithm for TCP. RFC 3517, Internet Engineering Task Force, Apr. 2003.

[11] M. C. Chan and R. Ramjee. TCP/IP performance over 3G wireless links with rate and delay variation. In *Proc. 8th Annual International Conference on Mobile Computing and Networking (MobiCom 2002)*, Sept. 2002.

[12] S. Floyd and E. Kohler. Profile for DCCP Congestion Control ID 2: TCP-like congestion control. Internet-Draft draft-ietf-dccp-ccid2-02, Internet Engineering Task Force, May 2003. Work in progress.

[13] S. Floyd and E. Kohler. Tcp friendly rate control (tfrc): the small-packet (sp) variant. Internet-Draft draft-ietf-dccp-tfrc-voip-04, Internet Engineering Task Force, Jan. 2006. Work in progress.

[14] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. In *Proc. SIGCOMM 2000*, Aug. 2000.

[15] S. Floyd, E. Kohler, and J. Padhye. Profile for DCCP Congestion Control ID 3: TFRC congestion control. Internet-Draft draft-ietf-dccp-ccid3-02, Internet Engineering Task Force, May 2003. Work in progress.

[16] A. R. H. Schulzrinne and R. Lanphier. Real time streaming protocol (rtsp). RFC 2326, Internet Engineering Task Force, Apr. 1998.

[17] R. F. H. Schulzrinne, S. Casner and V. Jacobson. Rtp: A transport protocol for real-time applications. RFC 3550, Internet Engineering Task Force, July 2003.

[18] F. Hammer, P. Reichl, T. Nordström, and G. Kubin. Corrupted speech data considered useful. In *Proc. 1st ISCA Tutorial and Research Workshop on Auditory Quality of Systems*, Apr. 2003.

[19] C. B. J. Widmer and J.-Y. L. Boudec. Congestion Control for Flows with Variable Packet Size. *ACM Computer Communication Review*, 34(2), 2004.

[20] V. Jacobson. Congestion avoidance and control. In *SIGCOMM 1988*, Aug. 1988.

[21] E. Kohler, M. Handley, S. Floyd, and J. Padhye. Datagram Congestion Control Protocol (DCCP). Internet-Draft draft-ietf-dccp-spec-02, Internet Engineering Task Force, May 2003. Work in progress.

[22] B. Landfeldt, T. Larsson, Y. Ismailov, and A. Seneviratne. SLM, a framework for session layer mobility management. In *Proc. IEEE ICCCN'99*, Boston, Massachusetts, 1999.

[23] L.-A. Larzon, M. Degermark, S. Pink, L.-E. Jonsson, and G. Fairhurst. The lightweight user datagram protocol (UDP-Lite). RFC 3828, Internet Engineering Task Force, July 2004.

[24] R. Ludwig and R. H. Katz. The Eifel algorithm: Making TCP robust against spurious retransmissions. *ACM Computer Communication Review*, Jan. 2000.

[25] A. Medina, M. Allman, and S. Floyd. Measuring the evolution of transport protocols in the Internet. *ACM Computer Communication Review*, 35(2):37–52, Apr. 2005.

[26] R. T. Morris. A weakness in the 4.2BSD Unix TCP/IP software. Computer Science Technical Report 117, AT&T Bell Laboratories, Feb. 1985.

[27] B. Mukherjee and T. Brecht. Time-lined TCP for the TCP-friendly delivery of streaming media. In *Proc. 8th International Conference on Network Protocols (ICNP '01)*, pages 165–176, Nov. 2000.

[28] J. Padhye and S. Floyd. On inferring TCP behavior. In *Proc. SIGCOMM 2001*, pages 287–298, Aug. 2001.

[29] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz. Known TCP implementation problems. RFC 2525, Internet Engineering Task Force, Mar. 1999.

[30] R. Stewart et al. Stream Control Transmission Protocol. RFC 2960, Internet Engineering Task Force, Oct. 2000.

[31] K. K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ECN) to IP. RFC 3168, Internet Engineering Task Force, Sept. 2001.

[32] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP congestion control with a misbehaving receiver. *ACM Computer Communication Review*, 29(5):71–78, Oct. 1999.

[33] A. Singh, A. Konrad, and A. D. Joseph. Performance evaluation of UDP Lite for cellular video. In *Proc. NOSSDAV 2001*, June 2001.

[34] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. 6th Annual International Conference on Mobile Computing and Networking (MobiCom 2000)*, Aug. 2000.

[35] A. C. Snoeren, H. Balakrishnan, and M. F. Kaashoek. Reconsidering internet mobility. In *Proc. 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Elmau/Oberbayern, Germany, May 2001.

[36] N. Spring, D. Wetherall, and D. Ely. Robust explicit congestion notification (ECN) signaling with nonces. RFC 3540, Internet Engineering Task Force, June 2003.

[37] S. Vanit-Anunchai, J. Billington, and T. Kongprakaiwoot. Discovering chatter and incompleteness in the Datagram Congestion Control Protocol. In *Proc. 25th IFIP International Conference on Formal Techniques for Networked and Distributed Systems*, Taipei, Taiwan, Oct. 2005.

[38] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of Internet flow rates. In *Proc. ACM SIGCOMM 2002 Conference: Applications, Technologies, Architectures, and Protocols for Computer Communication*, Pittsburgh, Pennsylvania, Aug. 2002.