Designing DCCP: Congestion Control Without Reliability

Eddie Kohler
UCLA
kohler@cs.ucla.edu

Mark Handley
University College London
M.Handley@cs.ucl.ac.uk

Sally Floyd
ICSI Center for Internet Research
floyd@icir.org

ABSTRACT

Fast-growing Internet applications like streaming media and telephony prefer timeliness to reliability, making TCP a poor fit. Unfortunately, UDP, the natural alternative, lacks congestion control. High-bandwidth UDP applications must implement congestion control themselves—a difficult task—or risk rendering congested networks unusable. We set out to ease the safe deployment of these applications by designing a *congestion-controlled unreliable transport protocol*. The outcome, the Datagram Congestion Control Protocol or DCCP, adds to a UDP-like foundation the minimum mechanisms necessary to support congestion control. We thought those mechanisms would resemble TCP's, but without reliability and, especially, cumulative acknowledgements, we had 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 how TCP's reliable bytestream semantics intertwine with its other mechanisms, including congestion control.

Categories and Subject Descriptors:
C.2.2 [**Computer-Communication Networks**]: Network Protocols; C.2.6 [**Computer-Communication Networks**]: Internetworking—*Standards (e.g., TCP/IP)*

General Terms: Design, Standardization

Keywords: DCCP, congestion control, transport protocols, unreliable transfer, streaming media, Internet telephony, TCP

1 INTRODUCTION

Selecting the right set of functionality for 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 the original functional decomposition looks remarkably good even 25 years later. The key omission from both TCP and IP was congestion control, which was retrofitted to TCP, the main bandwidth consumer, in 1988 [22]. Protocols other than TCP were appropriately left alone: TCP congestion control curbs the bandwidth usage of long-lived sessions, such as file transfers, and is bound up with TCP's flow control and reliable bytestream semantics; the TCP congestion control mechanisms are thus irrelevant for connectionless, unreliable applications such as DNS over UDP.

However, recent years have seen a large increase in applications using UDP for long-lived flows. These applications, which include streaming media, 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. By the time the old packet arrived, it would have been 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. An unreliable protocol is clearly more like what these applications want.

Applications generally do *not* want to implement TCP-friendly congestion control themselves. This is not only because congestion control can constrain performance, but also because properly implementing congestion control is very hard, as the long history of buggy TCP implementations makes clear [33, 34]. 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. A modular congestion control framework would also make it easier to develop new applications, and to deploy congestion control advances across many applications at once.

After analyzing several alternatives [17], and motivated mostly by keeping the basic API as simple as UDP's, we set out to design a new transport protocol providing a congestion-controlled flow of unreliable datagrams. The goal was a simple, minimal protocol upon which other higher-level protocols could be built—UDP, plus just those mechanisms necessary to support congestion control. The result, the Datagram Congestion Control Protocol (DCCP) [14, 18, 24], is currently an IETF Proposed Standard.

We expected the design process to run smoothly: after all, unreliability is simpler to provide than reliability, so surely unreliable congestion control would be no harder to provide than reliable congestion control. That naive expectation was wrong, and a protocol that should have been simple to design was not so simple after all. The development process helped us appreciate the ways TCP's reliability, acknowledgement, flow control, and congestion control mechanisms intertwine into an apparently seamless whole. In particular, DCCP's lack of retransmissions and 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 we still believe that an unreliable protocol's simpler semantics form a better base for layering functionality. We therefore discuss many of the issues we faced in designing a modern transport protocol, including some that the TCP designers did not face as squarely, such as robustness against attack.

Much of the work described here was done at the International Computer Science Institute Center for Internet Research in Berkeley, California.

This material is based in part upon work supported by the National Science Foundation under Grant Nos. 0205519 and 0230921. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

SIGCOMM'06, September 11–15, 2006, Pisa, Italy.

Related Work In the early days of Internet multimedia the research community naturally assumed that congestion control would be an integral part of UDP applications, although much of this work targeted multicast [11, 29]. In the end, commercial software vendors focused on unicast and omitted congestion control. Recently, applications such as Skype [41] have started to perform coarse-grained congestion adaptation to allow the use of higher quality codecs when bandwidth permits, but not in a form that encourages interoperability.

Systems such as Time-lined TCP [32] retrofit some support for time-sensitive data onto TCP, but do so using a specific deadline-based policy. Real applications often have more complex policies. For example, application-level messages may have different levels of importance and there may be interdependencies between them, the canonical example being MPEG's key frames (I-frames) and incremental frames (B/P-frames).

SCTP supports multiple datagram streams in a single connection [46]. This improves timeliness for some applications, since missing packets from one stream do not delay packets from any other stream. Nevertheless, SCTP's reliability, like TCP's, can introduce arbitrary delay. A partial reliability extension, PR-SCTP [45], attempts to overcome this by allowing a sender to explicitly abandon outstanding messages. This requires at least a round-trip time; the suggested API resembles Time-lined TCP's.

Another approach is to provide congestion control at a layer below TCP or UDP, as with the Congestion Manager [3, 6]. While this may have benefits for TCP, the benefits for unreliable UDP applications are less clear. These applications must provide their own protocol mechanisms to detect and acknowledge losses. This information is then fed to the Congestion Manager, which determines when the application can send. The necessarily tight coupling between feedback style and the congestion control algorithm makes this module breakdown rather unnatural. For example, adding smoother rate-based algorithms such as TFRC [16] to the Congestion Manager (as an alternative to the basic abruptly-changing AIMD algorithm) would require different feedback from the receiver; this would then require a new kernel API to supply the necessary feedback to the new Congestion Manager module.

Related work on architectural and technical issues in the development of new transport protocols includes papers on SCTP, RTP [39], RTSP [38], and UDP-Lite [26]. A peripherally related body of research on the development of new congestion control mechanisms for high-bandwidth environments, or with more explicit feedback from routers, highlights the need to be flexible to accommodate future innovation.

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.

One of DCCP's target applications is *Internet telephony*. Interactive speech codecs act like constant-bit-rate sources, sending a fixed number of frames per second. Users are extremely sensitive to delay and quality fluctuation—even more so than to bursts of static—so retransmissions are often useless: the receiver will have passed the playback point before the retransmission arrives. Quick adaptation to available bandwidth is neither necessary nor desired; telephony demands a slower congestion response. The data rate is changed by adjusting the size of each compressed audio frame, either by adjusting codec parameters or by switching codecs altogether. At the extreme, some speech codecs can compress 20 ms of audio down to 64 bits of payload. (The packet rate, however, is harder to adjust, as buffering multiple frames per packet causes audible delay.) Such small payloads pressure the transport layer to reduce its own header overhead, which becomes a significant contributor to connection bandwidth. A codec may also save bandwidth by sending no data during the silence periods when no one is talking, but expects to immediately return to its full rate as soon as speech resumes. Many of these issues are common to *interactive videoconferencing* as well, although that involves much higher bandwidth.

Streaming media introduces a different set of tradeoffs. Unlike interactive media, several seconds of buffer can be used to mask some rate variation, but since users prefer temporary video artifacts to frequent rebuffering, even streaming media generally prefers timeliness to absolute reliability. Video encoding standards often lead to application datagrams of widely varying size. For example, MPEG's key frames are many times larger than its incremental frames. An encoder may thus generate packets at a fixed rate, but with orders-of-magnitude size variation.

Finally, *interactive games* use unreliable transport to communicate position information and the like. Since they can quickly make use of available bandwidth, games may prefer a TCP-like sawtooth congestion response to the slower response desired by multimedia.

Since retransmissions are not necessarily useful for these time-sensitive applications, they have a great deal to gain from the use of Explicit Congestion Notification [35], which lets congested routers mark packets instead of dropping them. However, ECN capability must only be turned on for flows that react to congestion, which requires a negotiation between the two endpoints to establish. Most of these applications currently use UDP, but UDP's lack of explicit connection setup and teardown presents unpleasant difficulties to network address translators and firewalls and complicates session establishment protocols such as SIP. Any new protocol should improve on UDP's friendliness to middleboxes.

2.1 Goals

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

1. Minimalism. We prefer a protocol minimal in both functionality and mechanism. Minimal *functionality* means that, in line with the end-to-end argument and prior successful transport protocols in the TCP/IP suite, DCCP should not provide functionality that can successfully be layered above it by the application or an intermediate library. This helped determine what to leave out of the protocol; for instance, applications can easily layer multiple streams of data over a single unreliable connection. Minimal *mechanism* means that DCCP's core protocol features should be few in number, but rich in implication. Rather than solve protocol problems one at a time, we prefer to design more general mechanisms, such as the details of sequence numbering, that can solve several problems at once. We intended to design a simple protocol, but there are many kinds of simplicity: minimal mechanism defines the type of simplicity we sought in DCCP. Minimal mechanism also helps us achieve a secondary goal, namely minimal (or at least small) *header size*. To be adopted for small-packet applications such as Internet telephony, DCCP headers should be reasonably compact even in the absence of header compression techniques. For example, eight bytes is unacceptable overhead for reporting a one-bit ECN Nonce. Header overhead isn't critical for well-connected hosts, but we want to support DCCP on ill-connected, low-powered devices such as cell phones.

2. Robustness. The network ecosystem has grown rich and strange since the basic TCP/IP protocols were designed. A modern protocol must behave robustly in the presence of attackers as well as network address translators, firewalls, and other middleboxes.

First, DCCP should be robust against data injection, connection closure, and denial-of-service attacks. Robustness does not, however, require cryptographic guarantees; as in TCP, we considered it sufficient to protect against third-party attacks *where the attacker cannot guess valid connection sequence numbers* [31]. If initial sequence numbers are chosen sufficiently randomly [8], 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, requiring a search for reasonable middle ground. Middlebox robustness and transparency led us to introduce explicit connection setup and teardown, which ease the implementation burden on firewalls and NATs, and required the disciplined separation of network-level information from transport information. For example, our mobility design never includes network addresses in packet payloads or cryptographically-signed data.

3. A framework for modern congestion control. DCCP should support many applications, including some whose needs differ radically from file transfer (telephony, streaming media). To attract developers, DCCP should aim to meet application needs as much as possible without grossly violating TCP friendliness. Clearly DCCP should support all the features of modern TCP congestion control, including selective acknowledgements, explicit congestion notification (ECN), acknowledgement verification, and so forth, as well as obvious extensions hard to port to TCP, such as congestion control of acknowledgements. More importantly, congestion control algorithms continue to evolve to better support application needs. DCCP should encourage this evolution. Applications can thus choose among varieties of congestion control: DCCP provides a *framework* for implementing congestion control, not a single fixed algorithm. Currently, the choice is between TCP-like, whose sawtooth rates quickly utilize available bandwidth, and TFRC [16], which achieves a steadier long-term rate. In future, DCCP will support experimentation with new congestion control mechanisms, from low-speed TFRC variants to more radical changes such as XCP [23]. Each of these variants may require different acknowledgement mechanisms; for instance, TFRC's acknowledgements are much more parsimonious than TCP's. Thus, DCCP supports a range of acknowledgement types, depending on the selected congestion control method.

Another aspect concerns challenging links where loss and corruption unrelated to congestion are common, such as cellular and wireless technologies. Although there is no wide agreement on how non-congestion loss and corruption should affect send rates, DCCP should allow endpoints to declare when appropriate that packets were lost for reasons unrelated to network congestion, and even to declare that delivery of corrupt data is preferred to loss.

4. Self-sufficiency. 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. DCCP receivers must detect congestion events without application intervention; DCCP senders must calculate and enforce fair sending rates without application cooperation. Furthermore, congestion control parameters must be negotiated in-band.

5. Support timing–reliability tradeoffs. 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 might be preferred to video, for example), or newer packets preferred to older ones. 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.

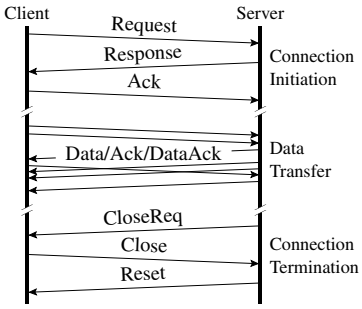


Figure 1: DCCP packet exchange overview.

2.2 Deliberate omissions

Any design is determined as much by what is left out as by what is included. During the lengthy DCCP design process, many suggestions were made to add functionality; most did not make the cut. In some cases it is interesting to note why not.

Flow control. In a reliable protocol it makes no sense to transmit packets that the receiver may discard. However, timing-critical applications may, under some circumstances, be unable to avoid doing so. Receivers may prefer to drop old data from their buffers in favor of new data as it arrives, or may prefer an application-specific policy difficult to express at the transport level. Flow control is also nontrivial to get right: likely-mistaken flow control limits have been observed to lower peak transfer rates [1, 48]. Thus, we decided that DCCP should not impose any flow control limitation separate from congestion control. This essentially extends support for timing–reliability tradeoffs to its logical endpoint. Of course, optional flow control could easily be layered on top of DCCP if desired.

Selective reliability. Prioritizing timeliness over reliability does not preclude retransmitting data, so long as the retransmissions reach the receiver in time. Transport-layer selective reliability might be convenient for applications, but we've found no obviously preferable API for identifying those datagrams that should be retransmitted; retransmission deadlines [32], maximum retransmission counts, and buffer-based strategies all have advantages and disadvantages. Since retransmissions are easily layered above DCCP, selective reliability was left out of the protocol itself for now.

Streams. SCTP [46] provides applications with a *stream* abstraction: sub-connection flows with independent sequence spaces. The benefit is that head-of-line blocking between streams is eliminated. For an unreliable protocol, though, there is no blocking problem, as neither reliable nor in-order delivery is guaranteed. It is trivial to layer streams over DCCP where they are required.

Multicast. It would have been nice to support multi-party delivery in DCCP, but there doesn't appear to be any simple common ground between the different possible uses of multicast, let alone between unicast and multicast. None of the main DCCP mechanisms, be it connection setup, acknowledgements, or even congestion control, apply naturally to multicast, and even among multicast applications one size does not fit all [21]. We resisted the temptation to generalize beyond what we believed we could do well.

3 DCCP Overview

DCCP 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. Since this field is 8 bits long, 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. Many packet types require additional information after the generic header, but before options begin; this design choice avoids cluttering the universal header with infrequently-used fields. Even the acknowledgement number is optional, potentially reducing header overhead for unidirectional flows of data. There are no equivalents to TCP's receive window and urgent pointer fields or its PUSH and URG flags, and TCP has no equivalent to CCVal (Section 6.2) or CsCov/Checksum Coverage (Section 6.5). Sequence and acknowledgement numbers are 48 bits long, although some packet types also permit a compact form to be used (see Section 4.5).

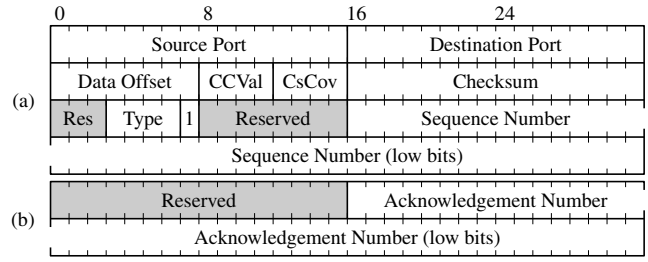


Figure 2: DCCP packet header. The generic header (a) begins every DCCP datagram. Individual packet types may add additional information, such as (b) an acknowledgement number. The packet header is followed by DCCP options, then payload; payload starts Data Offset words into the datagram.

4 Sequence Numbers

DCCP's congestion control methods are modularly separated from its core, allowing each application to choose a method it prefers. The core itself is largely focused on connection management—setup, teardown, synchronization, feature negotiation, and so forth.

The simplicity of this core functionality turned out to be a distinctly mixed blessing. TCP, for example, is able to simplify some aspects of connection management by leveraging the very semantics of reliability that it aims to provide. TCP's flow control means that two live endpoints always remain synchronized, and TCP's reliability means a single cumulative acknowledgement number suffices to describe a stream's state. More generally, TCP combines reliability, conciseness of acknowledgement, and bytestream semantics in a tightly unified whole; when we tried to separate those properties, its mechanisms fell apart. Sometimes the solutions we developed in response seem as simple as TCP's and sometimes they don't, but they are almost always different.

DCCP's core connection management features all depend on the most fundamental tool available, namely *sequence numbers*. We now know to consider sequence numbers carefully: seemingly small changes to sequence number semantics have far-reaching effects, changing everything up to the protocol state machine. The interlocking issues surrounding sequence numbers collectively form the most surprising source of complexity in DCCP's design, so we explore them in some depth.

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.

A cumulative ackno indicates that all sequence numbers up to, but not including, that ackno have been received. The receiver guarantees that, absent a crash or application intervention, it will deliver the corresponding 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, a more explicit approach implemented by TCP selective acknowledgements (SACK) [10].

TCP 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 [37]. 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, needs its own acknowledgement mechanism. Each timestamp option contains an acknowledgement; a TCP header bit (CWR) acknowledges ECN congestion reports; support for optional features is acknowledged via options like SACK-Permitted.

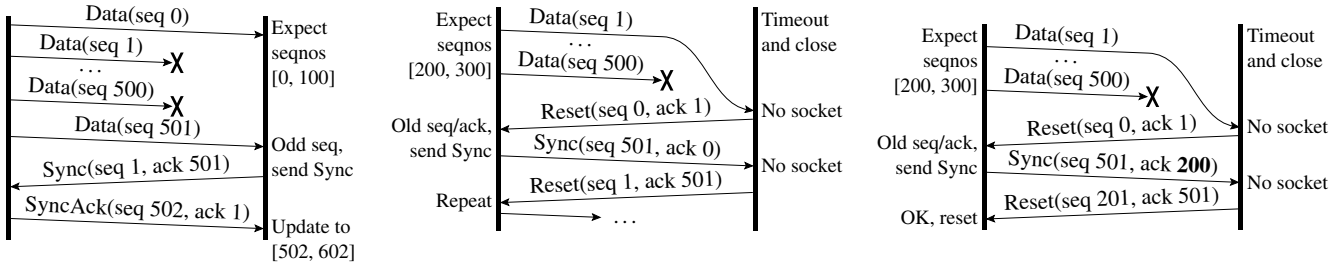
Pure acknowledgements, which contain neither data nor SYN or FIN bits, do not occupy sequence space, and thus cannot be acknowledged conventionally. As a result, TCP cannot easily evaluate the loss rate for pure acknowledgements or detect or react to reverse-path congestion, except as far as high acknowledgement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

4.2 DCCP sequence numbers

DCCP must be able to detect loss without application support. Inevitably, then, DCCP headers must include sequence numbers. Those sequence numbers should measure datagrams, not bytes, since in accordance with the principles of Application Layer Framing [13], unreliable applications generally send and receive datagrams rather than portions of a byte stream. This also simplifies the expression of congestion control algorithms, which generally work in units of packets. (Some care is required to calculate congestion control using the average packet size.)

What, though, should be done with packets that don't carry application data? DCCP's goals include applying congestion control to acknowledgements, negotiating congestion control features in band, and supporting explicit connection setup and teardown. The first goal requires detecting acknowledgement loss; the second requires acknowledging each feature negotiation. A single minimalist choice, motivated by TCP's inclusion of SYN and FIN in sequence space, seemed to address all three goals 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 example, a single sequence space now contains both data packets and acknowledgements. Often this should be separated: TCP does not reduce a sender's rate when an acknowledgement it sends is lost, so neither should DCCP.) The obvious TCP-like choice would have been to assign pure acknowledgements the same sequence numbers as preceding data packets; only connection handshakes and



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

Figure 3: Recovering synchronization after bursts of loss.

data would gain new sequence numbers. Of course, feature negotiation and connection synchronization would then require ad hoc acknowledgement mechanisms. Another alternative would be to introduce a secondary sequence number space for non-data packets. In the end, though, we believe that despite its warts, the minimalist path we chose is as simple as or simpler than these alternatives.

Most DCCP packets carry an acknowledgement number as well as a sequence number. This led to another critical design decision: To which packet should the ackno correspond? Cumulative acknowledgements don't make sense in an unreliable protocol where the transport layer never retransmits data. DCCP's ackno thus reports the *latest packet received*, rather than the earliest 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 congestion control-specific options provide information about packets preceding the ackno. The most detailed option, Ack Vector, reports exactly which packets were received, and exactly which packets were received ECN-marked, using a run-length-encoded byte array; each Ack Vector byte represents up to 64 packets.

4.3 Synchronization

When 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 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. Thus, large bursts of loss can force endpoints out of sync, a problem surprisingly difficult to solve.

We cannot eliminate expected-sequence-number windows, as they are the main line of defense protecting connections from attack (see Section 4.6). Instead, DCCP supports *explicit* synchronization. An endpoint receiving an unexpected sequence or acknowledgement number sends a Sync packet asking its partner to validate that sequence number. (TCP in this situation would send a reset.) The other endpoint processes the Sync and replies 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.

Some early versions of this mechanism synchronized using existing packet types, namely pure acknowledgements. However, *mutually* unsynchronized endpoints can never resync in such a design, as there is no way to distinguish normal out-of-sync 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. 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, and, since even probe packets occupy sequence space, 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, although subtlety is still required to deal with half-open connections.

4.4 Acknowledgements

A TCP acknowledgement requires only a 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. Furthermore, some types of acknowledgements use bounded state, and thus never need to be acknowledged.

Unreliability also affects the semantics of acknowledgement. In DCCP, an acknowledgement *never* guarantees that a packet's data

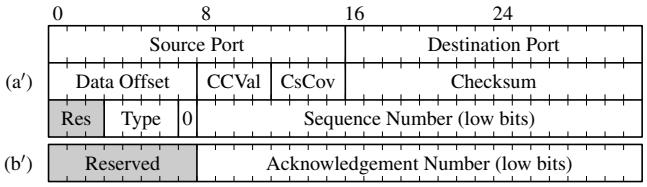


Figure 4: DCCP header with short sequence numbers. See also Fig. 2.

will be delivered to the application. This supports trading off timeliness against reliability (Goal 5). 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. It is desirable for the application, as part of the timeliness–reliability tradeoff, to be able to drop the old data.

For many reasons, however, this data should have been acknowledged already. Acknowledging packets only on application delivery would distort round-trip time measurements and unacceptably delay option processing; acknowledgement options should, for congestion control purposes, report only losses and marks that happened in the network proper. To avoid muddying the semantics, we separate these concerns at the expense of a little efficiency. DCCP acknos and acknowledgement options 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 option called Data Dropped indicates when an acknowledged packet's data was not delivered—for example, when that data was dropped in the receive buffer.

4.5 Sequence number length

How big should the sequence space be? Short sequence numbers lead to smaller headers, less bandwidth, and less endpoint state. On the other hand, they wrap more frequently—that is, long-lived connections must quickly reuse sequence numbers, running the risk that old delayed packets might be accepted as new—and 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). Despite this, DCCP originally used 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, 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 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, in under 2 minutes).

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

The solution, once found, was relatively clean. Although DCCP sequence numbers are 48 bits long, some packet types may leave off the upper 24 bits (Figure 4). The receiver will infer those bits' values using an expected 48-bit sequence number. The following procedure takes a 24-bit value 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. But data and acknowledgement packets—exactly those packets that will make up the bulk of the connection—may, if the endpoints approve, 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.

4.6 Robustness against attack

Robustness against 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 [31]. Applications desiring protection against snooping attacks must use some form of cryptography, such as IPsec 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 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/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_1W_2N/L^2. 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, recently-publicized attack with RST packets is somewhat easier to defend against.) Recent measurements report a median advertised window of approximately 32 kB [30]; with $W = 32768$ bytes, 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 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 $W = 2000$ packets (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 $W = 100$ packets, 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).

Unless we are careful, though, data injection might cause connection reset. For example, certain invalid options might cause the receiver to reset the connection; an injected Data packet might include such an option. Several aspects of the protocol were modified to prevent this kind of attack 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 Summary and discussion

Congestion control requires loss detection, which in turn requires sequence numbers. An unreliable protocol uses application data units, so DCCP sequence numbers name *packets* rather than bytes. Several reasons, including our preference for minimal mechanism, led us to assign *every packet* a new sequence number.

The semantics of acknowledgement are very different for an unreliable protocol than for TCP, as there is no succinct equivalent to TCP's cumulative ackno. DCCP acknowledges the *most recently received packet*. Options such as Ack Vector indicate precisely which packets have been received; some such options may grow without bound, requiring that *acknowledgements be acknowledged* from time to time.

Providing robustness via sequence number validity checks is harder for an unreliable protocol, since absent flow control, the two endpoints can get out of sync. DCCP thus provides an *explicit synchronization* mechanism. This has some advantages even over TCP's design, since unexpected events can trigger synchronization rather than connection reset.

Long sequence numbers are preferred to short ones, since they cleanly avoid wrapping issues and frustrate attack, but where space is at a premium, short sequence numbers can be *extended* to long ones on the fly. Care should be taken to ensure that any easily-attacked points in the protocol, such as opportunities for data injection, *cannot escalate* to denial-of-service attacks.

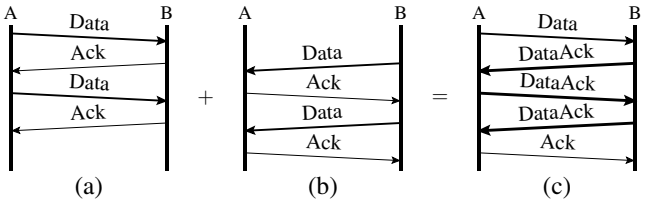


Figure 5: (a) An A-to-B half-connection and (b) a B-to-A half-connection combine into (c) a full connection with piggybacked data and acknowledgements.

Not all comparisons between TCP sequence numbers and DCCP-style unreliable, packet-oriented sequence numbers come out in favor of TCP. For example, TCP's bytestream 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 [27, 36].

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; after the initial connection setup, the client's packets are all acknowledgements.

TCP devolves naturally into unidirectional communication. Since TCP acknowledgements occupy no sequence space, it is neither useful nor possible to acknowledge them; since data retransmissions clean up old ack state, a unidirectional TCP flow in which all data has been acknowledged occupies minimal state on both endpoints. We aim for a similar property from DCCP: a DCCP connection with unidirectional data flow should spend little time, space, or bandwidth on the inactive direction. In a bidirectional DCCP connection, however, each endpoint may need to keep detailed SACK-like acknowledgement information about its partner's data packets. When data flows unidirectionally, this overhead is largely a waste for the inactive direction. If B is sending only acknowledgements to A, then A should acknowledge B's packets only as necessary to clear B's acknowledgement state; these acks-of-acks are minimal and need not contain detailed loss reports (Section 4.4).

To solve these issues cleanly, DCCP logically divides each connection into two *half-connections*. A 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 (Figure 5). The format for acknowledgements is determined by the governing half-connection's congestion control method, 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.

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

5.2 Feature negotiation

DCCP's connection endpoints must agree on a set of parameters, the most obvious of which is the choice of congestion control methods the connection should use. Both endpoints have capabilities—the mechanisms they implement—and application requirements—the mechanisms the application would prefer. Since the application cannot be relied upon to negotiate agreement, negotiation must take place in band. TCP has a similar problem, applying at least to ECN, SACK, window scaling, and timestamps, which it solves ad hoc with different options or bits in each case. The resulting complexity would only grow in an unreliable protocol. Therefore, in DCCP we built in a single general-purpose mechanism for reliably negotiating the values of *features*. A feature is simply a per-endpoint property on whose value both endpoints must agree. Examples include each half-connection's congestion control mechanism, and whether or not short sequence numbers are allowed.

Feature negotiation involves two option types: Change options open feature negotiation, and 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. A single exchange 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, generic reliable feature negotiation has allowed us to easily add additional functionality without needing to consider interactions between feature negotiation, congestion control, reliability, and the differing acknowledgement styles required by each congestion control mechanism.

5.3 Mobility and multihoming

Mobility and multihoming, which cut across the network and transport layers, are different from most functionality in that they cannot be layered on top of an unreliable protocol. Mobility could be implemented entirely at the network layer, as with Mobile IP, but choosing the transport layer has advantages [42]: the transport layer is naturally aware of address shifting, so its congestion control mechanism can respond appropriately, and transport-layer mobility avoids triangle routing issues. We were thus directed to develop a mobility and multihoming mechanism for DCCP.

Happily, mobility and multihoming are among 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 [42]. 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*, each of which may have different endpoint addresses, ports, sequence numbers, and even connection features, into a single *session* entity. This is done in the simplest possible way: to add a new address, an endpoint opens a new DCCP connection, including in its Request an option for attaching to an existing session. This means that most DCCP and middlebox 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 the application. Most transport state is unique per component connection, but all components in a session share a single socket. Data written 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 [25, 43], but thanks to unreliability, we can add multihoming support while simplifying the basic abstractions.

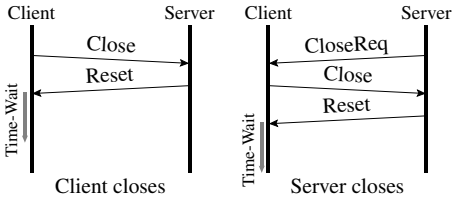


Figure 6: Shutdown handshakes push Time-Wait state to the client.

The mobility and multihoming mechanism also prevents connection hijacking, where an attacker moves one endpoint of a victim's connection to its own IP address. We reason that hijacking is fundamentally more serious than data injection or connection reset, so hijacking should be prevented *even when the attacker can passively snoop the connection*. Thus, the DCCP options that manage sessions are protected against forgery and replay by nonces and digital signatures. Of course, an on-path active attacker, such as a compromised router, can still hijack a connection with or without mobility.

5.4 Denial-of-service attacks

In a transport-level denial-of-service attack, an attacker tries to break a victim's network stack by overwhelming it with data or calculations. For example, the 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. Any modern transport protocol must be designed from the outset to resist such attacks, which may even involve changes to the design of the protocol state machine itself.

The basic strategy is to push state to the client whenever possible. In DCCP, for example, a server responding to a Request packet 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] and SCTP's initialization cookies [46], this lets the server avoid keeping any information about half-open connections; unlike SYN cookies, which were retrofitted, Init Cookies can encapsulate lots of state. Another state-holding issue occurs during connection shutdown where, as with TCP, Time-Wait state needs to remain at an endpoint for at least two minutes to prevent confusion in case the network delivers packets late. Unlike TCP, DCCP servers can shift Time-Wait state onto willing clients. This is accomplished by introducing asymmetry into the shutdown state machine. All DCCP connections end with a single Reset packet, and only the receiver of that Reset packet holds Time-Wait state. Normal connections end with a Close–Reset handshake, but the server (and only the server) can initiate shutdown with a CloseReq packet, which effectively asks the client to accept Time-Wait state (Figure 6).

DCCP also 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, the DCCP state machine itself and the explicit synchronization mechanism have both been engineered to resist blind reset attacks on existing connections.

5.5 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, [28]) model and an independently-developed colored Petri net (CPN) model from the University of South Australia [47]. 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 tool was simply shifting from reasoning via state diagrams to detailed pseudocode that defined how packets should be processed. The resulting precision revealed 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 Time-Wait 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 formal modeling was quite positive, especially combined with clear explanation in pseudocode. Next time, we would seek out modeling experts earlier in the design process.

6 Congestion Control

As a congestion control framework, DCCP gives the application a choice of congestion control mechanisms. Some applications might prefer TFRC congestion control, avoiding TCP's abrupt halving of the sending rate in response to congestion, while others might prefer a more aggressive TCP-like probing for available bandwidth. The choice is made via Congestion Control IDs (CCIDs), which name standardized congestion control mechanisms. A connection's CCIDs are negotiated at connection startup. This section describes the two CCIDs that have currently been developed, congestion control issues exposed by DCCP's target applications that remain to be solved, and more general problems relating to congestion control, including misbehaving receivers and non-congestion loss.

6.1 CCID 2: TCP-like Congestion Control

DCCP's CCID 2 provides a TCP-like congestion control mechanism, including the corresponding abrupt rate changes and ability to take advantage of rapid fluctuations in available bandwidth. CCID 2 acknowledgements use the 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].

One difference from TCP is CCID 2's reaction to reverse-path congestion. 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]. Modern protocols should ideally detect and act on reverse-path congestion. Thus, CCID 2 maintains a feature called Ack Ratio that controls the rough ratio of data packets per acknowledgement. TCP-like delayed-ack behavior is provided by the default Ack Ratio of two. As a CCID 2 sender detects lost acknowledgements, it manipulates the Ack Ratio so as to reduce the acknowledgement rate in a very roughly TCP-friendly way.

Ack Ratio is an integer. To reduce ack load, it is set to at least two for a congestion window of four or more packets. However, to ensure that feedback is sufficiently timely, it is capped at cwnd/2, rounded up. Within these constraints, the sender changes Ack Ratio as follows. Let R equal the current Ack Ratio.

- For each congestion window of data where at least one of the corresponding acks was lost or marked, R is doubled;
- For each $\text{cwnd}/(R^2 - R)$ consecutive congestion windows of data whose acks were not lost or marked, R is decreased by 1.

This second formula comes from wanting to increase the number of acks per congestion window, namely cwnd/R, by one for every congestion-free window that passes. However, since R is an integer, we instead find a k so that, after k congestion-free windows, $\text{cwnd}/R + k = \text{cwnd}/(R-1)$.

6.2 CCID 3: TFRC Congestion Control

TFRC congestion control in DCCP's CCID 3 uses a different approach. Instead of a congestion window, a TFRC sender uses a sending rate. The receiver sends feedback to the sender roughly once per round-trip time reporting the loss event rate it is currently observing. The sender uses this loss event rate to determine its sending rate; if no feedback is received for several round-trip times, the sender halves its rate.

This is reasonably straightforward, and does not require 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 event 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 event rate. Each loss interval contains a maximal tail of non-dropped, non-marked packets. The Loss Intervals option reports each tail's ECN nonce echo, allowing the sender to verify the acknowledgement; see Section 6.4 below. The receiver need never report more than the nine most recent Loss Intervals. Since this bounds acknowledgement state, CCID 3 acknowledgements need not be acknowledged. Loss Intervals resembles TCP's SACK option even more closely than does Ack Vector, except that unlike SACK, Loss Intervals can group several distinct losses into a single range representing a congestion event. This feedback information is substantially different from CCID 2's Ack Vector, but DCCP supports both mechanisms equally well. A less flexible protocol might have difficulties supporting future congestion control methods as the state of the art 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

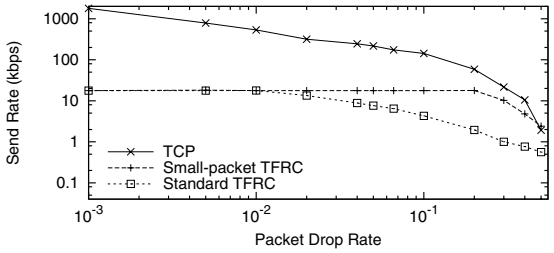


Figure 7: Send rate for given packet drop rates using TCP, standard TFRC, and small-packet TFRC. TCP uses bulk transfer and 1460-byte segments; TFRC uses 14-byte segments and has an application-limited maximum send rate of 12 kbps including headers.

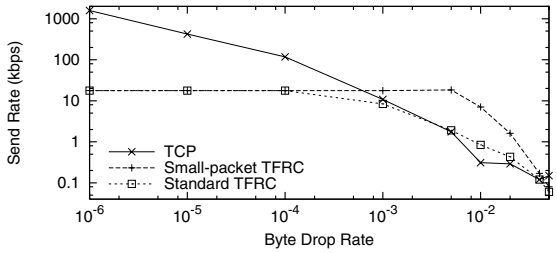


Figure 8: Send rate for given *byte* drop rates using TCP, standard TFRC, and small-packet TFRC. TCP uses bulk transfer and packet sizes that optimize send rate; TFRC parameters are as in Figure 7.

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.

6.3 Future congestion control issues

Many open issues remain for designing congestion control suitable for unreliable timing-critical applications. Examples of currently-problematic application desires include:

- Sending many small packets rather than fewer large ones.
- Rapid startup after idle periods, such as in interactive communication where parties speak in turn.
- Abrupt changes in application data rate due to codec artifacts, such as MPEG I-frames vs. B/P-frames.

We don't yet understand how far congestion control mechanisms for best-effort traffic can be pushed to deal with these application-level issues, or what the consequences might be for aggregate traffic if congestion control mechanisms are pushed too far. We expect DCCP to evolve as more is learned, and modular CCIDs facilitate this evolution. As a concrete example, we focus on the small packet issue, which casts light on the fundamental difficulties faced in designing a protocol that should work well for a wide range of applications in the face of immense diversity of network constraints.

For a fixed packet loss rate, a TCP connection that uses smaller packets will achieve a proportionally lower sending rate in bytes per second than one sending larger packets. However, TCP's bytestream semantics mean that it can generally assemble packets to be as large as possible. For unreliable applications, the story is rather different. Due to a combination of application-level framing and tight delay constraints, applications such as telephony and gaming may sometimes find it necessary to send frequent small packets. A good adaptive multi-rate CELP speech codec such as AMR [19] can achieve bitrates from 12 kbps down to less than 5 kbps. At 5.6 kbps, a 20 ms audio frame requires only 14 bytes. Interactive media must react to congestion primarily by adapting the packet *size*, keeping the rate constant; any additional latency would introduce audible artifacts into the playout stream.

So how should such a low-bandwidth, small-packet flow compete with a TCP flow sending 1500-byte packets? We initially hoped that standard TFRC would suit VoIP applications, but in practice it competes poorly because it factors packet size into its throughput equation. By default, a TFRC flow using small packets will achieve the same throughput as a TCP flow using the same small packet size and seeing the same loss rate. But most of the time TCP does not use small packets, so a TFRC VoIP session will lose out badly to a file transfer. See, for example, the simulation results in Figure 7: an application-limited standard TFRC flow reduces its send rate with increasing loss rates, even though it always sends far less data than would a large-packet TCP flow.

Given these sensible application requirements—and the applications' overall modest sending rates, in both packets and bytes per second—it made sense to design a TFRC variant allowing such a VoIP call to compete fairly with a large-packet TCP flow. Our small-packet TFRC variant [15] does precisely this by compensating for packet size. Figure 7 also shows that the small-packet TFRC variant competes fairly for bandwidth with a large-packet TCP flow.

Is small-packet TFRC safe to deploy? The issue is clouded by questions about the bottleneck links. While the bottleneck router's forwarding limitation is commonly link capacity in bytes per second, in some cases it may be router CPU cycles, which constrain the forwarding rate in *packets* per second. Even if link capacity is the bottleneck, the queue at the bottleneck router may be limited in packets or bytes. The former will give both small and large packets the same drop probability, whereas the latter will preferentially drop large packets. Furthermore, in some situations a flow might encounter *multiple* bottlenecks with different characteristics.

If the bottleneck is in packets per second, an adaptation that changes only the packet size while sending a constant packet rate serves no purpose. However, most modern routers can forward minimum sized packets at line speed, so it is probably reasonable to assume that changing the packet size is worthwhile. But does a small-packet TFRC flow in fact see the same loss rate as the large-packet TCP flow? If the bottleneck router manages its flow in bytes, then the small packets are already less likely to be dropped. Figure 8 shows the results of a simulation like Figure 7, but where each *byte* is dropped with some probability; a packet is dropped if any of its bytes are dropped. Here, *standard* TFRC competes fairly with TCP. The small-packet variant gets too much bandwidth at high byte drop rates, and can actually starve TCP flows in extreme circumstances.

Internet router behavior is simply not well specified, so there is no right answer for how congestion control should be designed. What then should DCCP do? The question of appropriate congestion control for small packet flows is still open. A pragmatic view is that applications will not choose between standard TFRC and small-packet TFRC, but rather between small-packet TFRC and no congestion control at all. If DCCP only offered standard TFRC, with the likelihood of behavior like that in Figure 7, many application writers would opt for a fixed-rate UDP flow. The small-packet variant is never worse for the network than this, and sometimes it is much better; and, importantly, it may work for the application.

6.4 Misbehaving receivers

Internet congestion control is voluntary in the sense that few, if any, routers actually enforce congestion control compliance. Un-

fortunately, 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 [37]. 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 generally tolerate loss to some degree, 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 [37].

DCCP, like TCP, uses the ECN Nonce for this purpose. The nonce encodes one bit of unpredictable information that is destroyed by loss or ECN marking [44]. 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; but in the presence of retransmission and partial retransmission, a TCP 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, if short sequence numbers are used, an attacker may be able to 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.

6.5 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 [20, 40]. 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 [12]. DCCP therefore follows the UDP-Lite protocol [26] 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 corrupt datagrams should be forwarded 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]. It isn't yet clear how one *should* respond to different types of loss—our current congestion control mechanisms treat corruption as they would treat ECN marking, that is, as congestion indications. However, protocols should at least allow receivers to distinguish between types of loss, allowing incremental deployment of alternative responses as experience is gained.

To enable this, 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.

6.6 Summary and discussion

DCCP was designed from the outset to support *modular* congestion control. In part, this is because the state of the art is still advancing, both algorithmically and in the proper response to *non-congestion loss*. Supporting this evolution in a transport protocol avoids the need to rewrite thousands of applications with every update to congestion control semantics. Furthermore, time-sensitive applications can have widely varying needs, as illustrated by small-packet TFRC. It seems unlikely that any one algorithm will suit them all, so allowing applications to choose the dynamics they prefer is essential for success.

This choice has consequences, though. Congestion control algorithms form a control loop; the dynamics of the algorithm and the nature of the feedback information are tightly coupled. Thus, selecting a specific algorithm also dictates the *acknowledgement format*.

The need to be robust in the face of attack also weighs heavily on the design of a modern protocol. Issues such as denial-of-service attacks, misbehaving receivers, and sequence number validity affect many small details. Robustness is actually very hard to get right—only formal modeling revealed some subtle flaws in our earlier designs. To expect every application designer to do such modeling is asking too much; when this work is done for a transport protocol a whole range of different applications can then reap the benefits.

7 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.

TCP manages such a beautifully integrated design for two main reasons. First, the bytestream abstraction is very simple. With the exception of the urgent pointer, TCP does not need to consider detailed application semantics. Second, TCP is able to bootstrap off its own reliability; for example, the cumulative acknowledgement in TCP serves many purposes, including reliability, liveness, flow control, and congestion control. An unreliable protocol has neither luxury, and there does not appear to be a simple unifying mechanism equivalent to the cumulative acknowledgement.

Nevertheless, it is possible to design a relatively simple protocol that robustly manages congestion-controlled connections with-

out reliability. Explicit synchronization and new acknowledgement formats even have some advantages over their TCP equivalents. Modular congestion control mechanisms make it possible to adapt congestion control within a fixed protocol framework as network and application constraints change. Robustness against attack is addressed in a more thorough way.

It is too early to tell whether DCCP will succeed in wide deployment. Only recently have implementations started to appear (in Linux and FreeBSD); NATs and firewalls do not yet understand it; no application yet uses DCCP as its primary transport. Because it was designed for applications, and with feedback from application designers, we hope and believe it will be useful anyway. Regardless, our design experience cast well-known issues of reliability and protocol design in what seemed to us a valuable new light.

Although it may not seem like it, we have deliberately avoided describing all the details of DCCP. The interested reader is referred to the specifications [14, 18, 24].

CUMULATIVE ACKNOWLEDGEMENTS

DCCP has benefitted from discussions with many people, including our early collaborator Jitendra Padhye, our working group chair Aaron Falk, and members of the DCCP, TSV, and MMUSIC Working Groups and the End-to-End Research Group. We lack room to name most individuals here—a full list appears in the specification [24]—but we especially thank Junwen Lai, who helped design the feature negotiation mechanism; Somsak Vanit-Anunchai, Jonathan Billington, and Tul Kongprakaiwoot for their CPN model [47]; and Andrei Gurtov and Arun Venkataramani. The first author thanks ICSI for continuing to provide a home away from home. Finally, we thank the anonymous reviewers for their helpful suggestions, including the one about the title of this section.

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. S. Rahul, and S. Seshan. An integrated congestion management architecture for Internet hosts. In *Proc. ACM SIGCOMM '99*, pages 175–187, 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] J.-C. Bolot, T. Turletti, and I. Wakeman. Scalable feedback control for multicast video distribution in the Internet. In *Proc. ACM SIGCOMM '94*, pages 58–67, Aug. 1994.

[12] M. C. Chan and R. Ramjee. TCP/IP performance over 3G wireless links with rate and delay variation. In *Proc. MobiCom 2002*, Sept. 2002.

[13] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proc. ACM SIGCOMM '90*, pages 200–208, Sept. 1990.

[14] S. Floyd and E. Kohler. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control. RFC 4341, Internet Engineering Task Force, Mar. 2006.

[15] S. Floyd and E. Kohler. TCP Friendly Rate Control (TFRC): the Small-Packet (SP) variant. Internet-Draft draft-ietf-dccp-tfrc-voip-05, Internet Engineering Task Force, Mar. 2006. Work in progress.

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

[17] S. Floyd, M. Handley, and E. Kohler. Problem statement for the Datagram Congestion Control Protocol (DCCP). RFC 4336, Internet Engineering Task Force, Mar. 2006.

[18] S. Floyd, E. Kohler, and J. Padhye. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC). RFC 4342, Internet Engineering Task Force, Mar. 2006.

[19] GSM 06.90. Adaptive Multi-Rate (AMR) speech transcoding.

[20] 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.

[21] M. Handley, S. Floyd, B. Whetten, R. Kermode, L. Vicisano, and M. Luby. The reliable multicast design space for bulk data transfer. RFC 2887, Internet Engineering Task Force, Aug. 2000.

[22] V. Jacobson. Congestion avoidance and control. In *Proc. ACM SIGCOMM '88*, Aug. 1988.

[23] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. In *Proc. ACM SIGCOMM 2002*, pages 89–102, Aug. 2002.

[24] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol. RFC 4340, Internet Engineering Task Force, Mar. 2006.

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

[26] 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.

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

[28] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. Wiley, 1999.

[29] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *Proc. ACM SIGCOMM '96*, pages 117–130, Aug. 1996.

[30] 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.

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

[32] B. Mukherjee and T. Brecht. Time-lined TCP for the TCP-friendly delivery of streaming media. In *Proc. ICNP 2000*, pages 165–176, Nov. 2000.

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

[34] 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.

[35] 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.

[36] P. Sarolahti, M. Kojo, and K. Raatikainen. F-RTO: An enhanced recovery algorithm for TCP retransmission timeouts. *ACM Computer Communication Review*, 33(2):51–63, Apr. 2003.

[37] 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.

[38] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326, Internet Engineering Task Force, Apr. 1998.

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

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

[41] Skype. Web page. `http://www.skype.com`.

[42] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. MobiCom 2000*, Aug. 2000.

[43] A. C. Snoeren, H. Balakrishnan, and M. F. Kaashoek. Reconsidering Internet mobility. In *Proc. HotOS-VIII*, May 2001.

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

[45] R. Stewart, M. Ramalho, Q. Xie, M. Tuexen, and P. Conrad. Stream Control Transmission Protocol (SCTP) Partial Reliability extension. RFC 3758, Internet Engineering Task Force, May 2004.

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

[47] 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*, Oct. 2005.

[48] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of Internet flow rates. In *Proc. ACM SIGCOMM 2002*, Aug. 2002.