Designing DCCP: Congestion Control Without Reliability

Eddie Kohler Mark Handley Sally Floyd

ICSI Center for Internet Research
{kohler, mjh, floyd}@icir.org

[1]

Abstract

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. Delay-sensitive applications, such as streaming media and telephony, prefer timeliness to reliability. These applications have historically used UDP and implemented their own congestion control mechanisms—a difficult task—or no congestion control at all. DCCP will make it easy to deploy these applications without risking congestion collapse. It aims to add to a UDP-like foundation the minimum mechanisms necessary to support congestion control, such as possibly-reliable transmission of acknowledgement information. This minimal design should make DCCP suitable as a building block for more advanced application semantics, such as selective reliability. We introduce and motivate the protocol and discuss some of its design principles. Those principles particularly shed light on the ways TCP's reliable byte-stream semantics influence its implementation of congestion control.

1 Introduction

Providing just the right set of functionality in a network protocol is a subtle art, 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 retro-fitted to TCP in 1988 [12]. Network protocols which did not use TCP were left to do their own thing, which was probably reasonable because TCP-like congestion control isn't appropriate for all applications.

Recent years have seen a large increase in applications using UDP for long-lived flows, and most of these applications have been reluctant to use congestion control. In part this may be because the application-designers don't have this on their list of priorities, and in part it may be because they don't have the necessary expertise to do it right.

Given the importance of congestion control to the correct functioning of the Internet, we decided that it was time to re-evaluate whether an alternative to UDP should be provided for applications for which TCP's semantics were inappropriate. The aim was to provide a simple minimal congestion control protocol upon which other higher-level protocols could be built. We called this the Datagram Congestion Control Protocol (DCCP) [13, 9, 10], and we expected it to be simple to provide an unreliable alternative to TCP. The issues turned out to be more complex than we expected.

In this paper we discuss our motivations for designing DCCP, for the particular set of functionality that DCCP eventually included, and some of the subtle issues involved in designing a congestion control protocol for unreliable flows.

2 Motivation

In the last few years there has been a steady growth of applications such as Internet telephony, streaming video and on-line games, that generate long-lived flows of UDP datagrams and share a preference for timeliness over reliability. For these applications, data not delivered within some deadline (typically a small number of round-trip times) will not be useful at the receiver. TCP can introduce arbitrary delay because of its reliability and in-order delivery requirements; thus, these applications use non-congestion-controlled UDP instead. This lack of congestion control poses a threat to the network: if these applications' usage continued to grow, the Internet might be at real risk of congestion collapse. Applications could implement their

[1] This material is based in part upon work supported by the National Science Foundation under Grant No. 0205519.

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.

own congestion control mechanisms on a case-by-case basis on top of UDP, and some already do. Implementing congestion control is difficult and error-prone, however, as the long history of buggy TCP implementations makes clear [16, 17], and new applications are unlikely to do it correctly on their own. We believe that a new transport protocol is needed, one that combines unreliable datagram delivery with built-in congestion control. This protocol would act as an enabling technology: new and existing applications could use it to easily transfer timely data without destabilizing the Internet.

2.1 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 [8] rather than strict TCP-like congestion control.[2]

- **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 [19] 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] TCP-Friendly Rate Control (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.

2.2 Design alternatives

Instead of designing a new transport protocol, we could use UDP and provide provide congestion control above or below it. Another alternative would be to modify an alternate transport protocol such as SCTP, which has some mechanisms for accomodating partial reliability.

2.2.1 Congestion control above UDP

It would seem unproductive to force the burden of implementing congestion control onto applications; congestion control is both difficult to implement and not part of most applications' core needs. While one could design and disseminate a user-level library of congestion control algorithms, a new transport protocol can do better in terms of ECN usage, middlebox traversal, features, and interoperability.

To allow the use of ECN by a user-level library, the UDP socket API would have to allow the application direct control over ECN fields in the IP header. This is somewhat risky, as it makes it rather too easy to send a flow that claims to react to ECN-marking when in fact it does not. In the absence of the widespread deployment of mechanisms in routers to detect flows that are using unreasonable bandwidth, we would not advise adding ECN capability to UDP sockets at this time.

A new transport protocol implemented in the kernel could almost certainly provide more speed and features than a generic UDP socket adapted for congestion control. Acknowledgement packets might be generated instantly, in the network interrupt for packet receipt; upcalls could be made exactly when a congestion control algorithm declared that a packet could be sent.

Finally, we believe that a transport protocol effectively achieves a level of interoperability higher than most libraries. For example, the transport protocol's generic specification makes it more accessible to embedded system designers.

2.2.2 Congestion control below UDP

A second possibility would be to provide congestion control for unreliable applications at a layer below UDP, with applications using UDP as their transport protocol, and with congestion feedback either at the application layer or at the layer below UDP. The Congestion Manager [2, 3] is an example of such an approach. Unfortunately, this does not necessarily provide access to multiple congestion

control mechanisms or address UDP's middlebox traversal issues, and approaches that rely on application-level feedback still push much of congestion control's complexity up to the application. We note that any new transport protocol could also use a Congestion Manager approach to share congestion state between flows using the same congestion control algorithm, if this were deemed to be desirable.

2.2.3 Congestion control at the transport layer

It would also be possible to modify TCP or SCTP to provide unreliable semantics, or modify RTP to provide congestion control. TCP seems particularly inappropriate, given its byte-stream semantics and reliance on cumulative acknowledgements; changing the semantics to this extent would significantly complicate TCP implementations and cause serious confusion at firewalls or monitoring systems.

SCTP [18] is a better match, as it was originally designed with packet-based semantics and out-of-order delivery in mind. However, the overlap with our requirements is partial at best; SCTP is hardly minimal, provides functionality that is unnecessary for many of the applications that might use DCCP, and does not currently allow the negotiation of different congestion control semantics.[3]

Adding congestion control to RTP [21] seems a reasonable option for audio/video applications. However, the UDP issues also mostly apply to RTP. In addition, RTP carries rather too many application semantics to really form a general-purpose building block for the future. Carrying RTP over DCCP seems a cleaner separation of functionality.

2.3 Mimimal DCCP requirements

Examining the success of TCP and UDP over the years, we observe that most of the applications they originally supported are of secondary importance today. Applications change, but these transport protocols are still useful because the services they provide are simple and general. There are few application-specific semantics in TCP, with the possible exception of the urgent pointer. Almost anything requiring reliable in-order delivery can be layered over TCP, and even the in-order delivery constraints are imposed more by the socket API than by the TCP protocol. What then would be the equivalent for a protocol providing an unreliable congestion-controlled packet stream?

Clearly such a protocol must provide UDP's demultiplexing and checksum functionality. Congestion control must take place in the context of a flow of packets, and we need a sequence number space to be able to discuss packet arrivals or losses. Congestion control also requires a feedback channel to convey congestion information back to the sender, including support for ECN. Strictly speaking, this is all that is required.

If we only provide the functionality above, we require out-of-band mechanisms for flow setup, port allocation, and for negotiation of any congestion control parameters. Is there any benefit to doing this out-of-band? Perhaps, because an out-of-band mechanism can satisfy its own timeliness and reliability constraints, but in today's Internet this seems to be outweighed by the difficulties of middlebox traversal for protocols that don't explicitly signal their own setup and teardown, and by the inconvenience and additional failure modes of using two protocols in parallel. Thus it seems wise to include connection setup and teardown functionality in such a protocol. As different applications have different requirements of congestion control, a mechanism for the negotiation of congestion control parameters is also needed.

To support the applications we currently envisage, there are many other features that might be desirable, such as packet-level FEC [11], selective reliability or limited retransmission, and support for multiple data streams in a flow. However all of these features can be supported just as efficiently over the top of a more simple protocol. We believe it is better not to include any functionality that is equally well provided at a higher layer, because this avoids inadvertently tailoring the protocol to the transient semantics of today's applications, which might prove useless baggage for those of tomorrow.

Functionality that cannot be layered over such a simple protocol is a short list: mobility and security.

Section 5 describes some of DCCP's security issues in detail. As for mobility, there are advantages to designing mobility into a transport protocol, rather than relying on other layers (Mobile IP or application-level connection restart). First, congestion control is naturally aware of address shifting, and can respond appropriately. Second, any security parameters associated with the normal flow processing can be leveraged in securing the move itself. Last, at least with IPv4, it avoids the triangle-routing caused by an endpoint being unable to assume IP mobility capability in the other party. For these and other reasons, DCCP includes a simple mobility mechanism; but even so, the authors are not completely convinced that this was the right choice.

[3] For a complete evaluation of SCTP for this purpose, see [7].

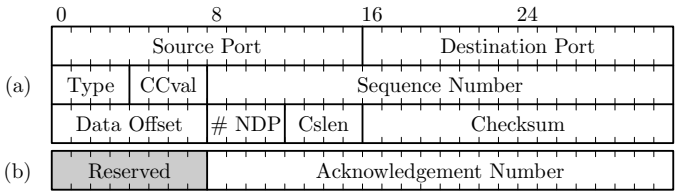

Figure 1—DCCP packet headers. The generic header (a) comes at the beginning of every DCCP datagram. Individual packet types may add additional information, such as (b) an acknowledgement number. In any case, the packet header is followed by space for options; any payload starts Data Offset words into the datagram.

3 Fundamentals and Framework

Our goal, given the motivations above, is to design a protocol that is lightweight and minimal, while providing a transport on which other mechanisms can be layered. The primary purpose is to support congestion control. Given that different applications have different requirements of congestion control, the protocol must support negotiation of congestion control mechanisms. As congestion control involves keeping state for the flow in the end-points, we need well-defined mechanisms to set up and cleanly tear down that state. And the pragmatics of deployment in today's Internet mean that NAT and firewall traversal must be taken into account.

This section describes DCCP's basic design choices and some consequences of its requirements. We present DCCP header structure, its feature negotiation mechanism, the problem of reliable acknowledgements on an unreliable connection, and justify its bidirectionality.

3.1 Packet structure

DCCP datagrams begin with a 12-byte generic header, showed in Figure 1. The four-bit Type field marks a difference from TCP: there are several (currently nine) different types of DCCP packet, rather than a single packet type whose meaning depends on a collection of flags. Compared to TCP, this design increases the number of packet types available (a four-bit field represents 16 types, not 4 flags), eliminates the possibility of confusing flag combinations ("Christmas tree packets"), and makes the smallest DCCP headers as small as possible (individual packet types can add information, such as an Acknowledgement Number, after the generic header).

DCCP has an option mechanism similar to TCP's—options are used for acknowledgement reporting and and parameter negotiation, for example—but its header allows much more option space: at most 1008 bytes, 25 times more than TCP. (Of course, we expect most packets will carry few options.)

3.2 Feature negotiation

At the beginning of a DCCP connection, the 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 mechanism the application would prefer), and capabilities and application requirements should both be able to influence the outcome. 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. DCCP, in contrast, provides a minimal set of options for negotiating the values of general *features*, where a feature is simply a value meant to be negotiated.

A "Change" option sent from A to B asks B to change its value for some feature. B can respond with either "Prefer", which says that it would prefer a different value, or "Confirm", which confirms that the feature's value has changed. Change and Prefer options are retransmitted until a response is received, making negotiation reliable. An endpoint that thinks a negotiation is taking too long may reset the connection.

Most features have values at both endpoints. Thus, a single packet sent from A to B might contain both "Change(f)" and "Confirm(f)" options, the first referring to the f feature located at B and the second referring to the f feature located at A. Feature negotiations for different features may take place in parallel.

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.

3.3 Reliable acknowledgements

TCP's acknowledgements are built on a cumulative acknowledgement field. Two acknowledgements with the same cumulative-ack convey the same information,[4] and TCP mechanisms like ack clocking and the retransmit and persist timers make acknowledgements reliable. The result is that a TCP receiver needs to keep only a fixed amount of acknowledgement state.

Ideally, a congestion control protocol for unreliable data would be as nicely integrated, efficient and safe, but

[4] Modulo SACK information, which is advisory.

the problem turns out to be significantly more complex. Firstly, cumulative acknowledgements have no meaning in an unreliable protocol: you never retransmit a packet. Secondly, depending on the congestion control mechanism, both ECN Nonce verification (Section 5.1) and application reporting may require that the receiver report exactly which packets it received. The result is that the acknowledgement state required at the receiver can grow without bound.

There is only one way to cut back on this state growth: the sender must occasionally acknowledge *one of the receiver's acknowledgements*. Once the receiver knows that one of its acks arrived at the sender, it can throw away the state that it sent in that ack. Thus, even though the protocol is unreliable, acknowledgements may need to be at least partially reliable.

There are several potential ways to do this:

- Treat acknowledgements like features, and use an explicit handshake to transfer acknowledgement information.
- Add an option to be used occasionally by the sender that means "I received feedback for everything up to my sequence number s".
- Make acknowledgements take up sequence number space. Then, when the sender reports that it received packet p, the receiver knows that p's acknowledgement options were received.

DCCP chooses this last approach as the cleanest and most general solution to the problem. (The handshake approach has high overhead, while an ad-hoc solution for acknowledgements wouldn't apply to anything else.) A DCCP sequence number is a 24-bit number that increases by one on every packet sent, including acknowledgements.

Per-packet sequence numbers have the happy side-effect that acknowledgements can finally be congestion controlled. In TCP, there is no way to congestion-control acknowledgements: since they're effectively equivalent, no one can tell how many were dropped by the network. In DCCP, even pure acks take up sequence space, so detecting reverse-path congestion is trivial. Some of the side effects of per-packet sequence numbers are less happy—see Section 5.2.1 on sequence number validity, for example—but overall, this design decision seems to have been a win.

3.4 Bidirectionality

DCCP, like TCP, provides a single bidirectional connection: both data and acknowledgements can flow in both directions.

This might seem less appropriate for DCCP than for TCP. In particular, with streaming media, the roles are very asymmetric. The server may transmit a huge live video feed to the client, using DCCP's congestion control to the fullest, while the client may have transmitted only a filename.

We originally considered making DCCP a unidirectional protocol, where all data flows from server to client except for an initial client-to-server request, which was not congestion controlled. In practice there are a number of reasons why this approach is undesirable. First, it's not clear when the client-to-server channel will cease. What if part of the request were dropped and the application needed to retransmit it, or what if the client wanted pause/play/rewind functionality? Retransmissions and further communication must be congestion controlled. Second, using two unidirectional connections for telephony would prevent acknowledgement information being piggybacked on data packets flowing in the same direction, and so would be rather inefficient. Finally, NATs and firewalls complicate the picture; their traversal is somewhat simplified if only a single connection is required (usually initiated from inside) for bidirectional communication.

We are still faced with the possibility of asymmetric functionality in a single connection, however. Our solution is to break a DCCP connection into two logical *half-connections*. Given a connection between A and B, one half connection comprises data packets from A to B and acknowledgement data from B to A, and the other comprises data from B to A and acks from A to B; see Figure 2. Feature negotiation for the two half-connections is completely independent, and may happen simultaneously. For example, the two half-connections may use different congestion control mechanisms.

This adds some complexity to the protocol. For example, two congestion control mechanisms may need to cooperate to produce a single piggybacked data-plus-ack packet, and a quiescence mechanism is required to reduce packet overhead in the common case of unidirectional communication (see below). Half-connections nevertheless have significant benefits in flexibility of use.

3.5 Quiescence

Again, we expect that many applications will use DCCP essentially unidirectionally, with most data flowing from server to client. A *quiescence* mechanism ensures that the protocol can handle unidirectional communication without unreasonable overhead. When one endpoint, say A,

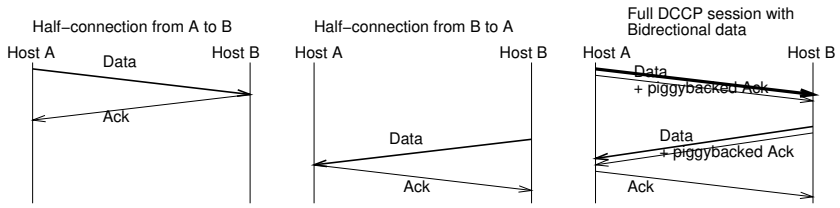

Figure 2—The two half-connections that make up a DCCP connection.

stops sending data for some amount of time (currently 2 RTTs, with some other constraints), the other endpoint, B, detects that A has gone quiescent and shifts to a unidirectional pattern of communication. This affects, mostly, the acknowledgements it sends. The quiescent endpoint sends only acknowledgements by definition. While A might demand Ack Vector for feedback on its data packets, it likely will not require such precise feedback—or, perhaps, any feedback at all—for its acknowledgements. Therefore, B, the non-quiescent endpoint, will limit the acknowledgements it sends to exactly those acks-of-acks required for its congestion control mechanism.

4 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.

4.1 TCP congestion control

Before we describe DCCP's congestion control mechanisms, this section provides a quick overview of TCP's, to serve as a point of comparison.

TCP's congestion control mechanisms involve an intertwining of flow control, congestion control, reliability, and in-order delivery. For example, the question of *what* to send is conflated with the question of *when* next to send a packet. TCP's flow control mechanisms are also closely related to TCP's use of in-order delivery. The TCP receiver tells the sender the receiver's advertised window and the lowest byte not yet delivered to the application; this restricts what the sender is able to send.

TCP's congestion control framework is a natural outgrowth of the flow control framework, with the sender maintaining a congestion window, and cumulative acknowledgements informing the sender that old data has left the network.

There is no need for a TCP receiver to check whether its acknowledgements arrived at the sender.[5] First, the information in TCP acknowledgements is cumulative (ignoring SACK information, which is advisory): subsequent acknowledgements will provide the same or newer information. More importantly, when too many TCP acknowledgement packets are lost, TCP's receive and congestion windows mean the sender will quickly stop sending: new acknowledgements are required to move the window forward.

4.2 CCID 2: TCP-like Congestion Control

DCCP provides a TCP-like congestion control mechanism, labeled CCID 2. However, because this operates in the context of an unreliable transfer, much of the congestion control framework differs from that of TCP.

TCP's stringent flow-control mechanism is not needed with DCCP. If packet n has been received but not yet read by the application, and packet $n + m$ then arrives, DCCP can choose to drop packet n from its receive buffer and use the buffer space to store the more recent packet.

DCCP's TCP-like congestion control still uses the sender's congestion window to limit the number of unacknowledged packets outstanding in the network, but it cannot use a cumulative acknowledgement field to control this. Thus some other mechanism is needed to ensure that if packets

[5]The exception is ECN, where the sender's Congestion Window Reduced (CWR) flag acts to confirm that an acknowledgement with ECN Congestion Experienced (ECE) set was received.

are lost, the sender halves its sending rate appropriately.

There are a number of ways that this could be accomplished in an unreliable transfer. One possibility would be to use a mechanism similar to TCP's SACK option [6]; the receiver sends acknowledgements of packets received, repeated in several subsequent packets for robustness. If the sender did not hear from the receiver that all of the packets in a window of data were received, along with reports of the ECN status of those packets, then the sender would be compelled to halve its congestion window. This would be safe, in that the sender would always back off when it should, but it could also result in unnecessary reductions of the congestion window if so many acknowledgements were lost that the sender did not hear about the receipt of some packet.

The alternate mechanism used by DCCP is to *reliably* transmit acknowledgement information from the receiver to the sender, using an Ack Vector and acks-of-acks. Essentially, the receiver keeps telling the sender that packet k has been received until the sender acknowledges some receiver message that included an Ack Vector covering k. The Ack Vector describes exactly which packets have been received, and whether those packets were ECN-marked in the network. The congestion control algorithm used to react to this information closely resembles that for SACK TCP.

A second vector may optionally be used to inform the sending application of packet payloads dropped by the receiving endpoint—because the receive buffer was full, for example. This ensures there is no ambiguity regarding metadata receipt; if a packet was reported received in the primary Ack Vector then the metadata will have been processed, even if the packet's payload was subsequently discarded. Section 5.1.1 further describes this issue.

One of the limitations of TCP is that there is no congestion control for the acknowledgements sent by the receiver to the sender. Ack congestion control can be useful any time there is congestion on the reverse path, but is particularly important for bandwidth-asymmetric networks or packet radio subnetworks [1]. DCCP, unlike TCP, can detect reverse-path congestion using per-packet sequence numbers, and respond to it as appropriate. In CCID 2, the DCCP sender responds by modifying the Ack Ratio, which controls the rate of the acknowledgement stream from the receiver. The algorithm used to set the Ack Ratio gives an ack sending rate that is very roughly TCP-friendly.

4.3 CCID 3: TFRC Congestion Control

TFRC congestion control in DCCP's CCID 3 uses a completely different approach than the TCP-like congestion control in CCID 2. 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. Complications do arise if the sender wants to verify for itself that the receiver's reported loss rate is accurate. For this, the receiver reports the ECN Nonce Sum for the long sequence of packets reported received and not ECN-marked at the end of each loss interval.

A key point to note is 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.

Of course, a sending application might want to know exactly which packets were received by the receiver for its own reasons. In these cases, a CCID 3 half-connection can additionally include Ack Vectors and acks-of-acks, as in CCID 2.

4.4 Partial checksums

The design of DCCP allows the DCCP checksum to cover all of the packet, just the DCCP header, or both the DCCP header and some number of bytes from the payload. This follows the proposal for a partial checksum in UDP-Lite [14]; the motivation for partial checksums is that some applications (e.g., voice and video) would prefer to have partially damaged payloads delivered rather than discarded by the network. However, because DCCP is a congestion-controlled transport protocol, some of the design issues for adding partial checksums to DCCP are more complex (and perhaps more compelling) that the issues with UDP.

Because of DCCP's use of end-to-end congestion control, if DCCP only allowed a checksum that covered the packet's payload, then a bit error in the payload would result in a dropped packet, and this packet drop would necessarily (but possibly incorrectly) be treated by DCCP as an

indication of congestion in the network. However, a corrupted packet is generally not an indication of congestion, and it is unnecessary for the transport protocol to reduce its sending rate in response to a single packet with a corrupted payload. To address this issue of an inappropriate congestion control response to a packet with a corrupted payload, DCCP also allows the use of separate checksums for the header and payload. This allows DCCP to detect payload corruption, but still not to mistake corruption for network congestion.

We note that the usefulness of partial checksums remains to be determined. For example, if the link-layer CRC on noisy links always discarded corrupted packets, this would limit the usefulness of partial checksums. In addition, as discussed in [13], partial checksums do not coexist well with IP-level authentication mechanisms such as IPsec AH, which cover the entire packet with a cryptographic hash. On balance, however, our belief is that DCCP partial checksums have the potential to enable some future uses that would otherwise be difficult. As the cost and complexity of supporting them is small, it seems worth including them at this time.

5 Guarding Against Misbehavior

DCCP includes tools that endpoints need to prevent misbehavior. Specifically, it addresses misbehaving-receiver attacks, where a greedy endpoint tries to get more than its fair share of network bandwidth; hijacking attacks, where a man-in-the-middle takes over a connection; and denial-of-service attacks, where a malicious or simply broken partner sends useless messages that nevertheless take up CPU or memory resources. Our goal was to make DCCP at least as safe against misbehavior as a state-of-the-art modern TCP implementation.

The issues encountered in preventing misbehavior deserve description, in particular because several issues were more difficult than in TCP because of DCCP's unreliability. Nevertheless, DCCP seems at least as protected against misbehavior as TCP.

5.1 Misbehaving-receiver attacks

Internet congestion control is voluntary today, in the sense that few, if any, routers actually enforce congestion control compliance on flows. 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, avoiding congestion control responses from the sender. Receivers could also acknowledge data before it arrives [20]. In TCP, the risk of a future packet loss can deter misbehaving receivers from this action, since missing data violates TCP's reliable, in-order semantics and must often be handled by the application. However, DCCP applications must tolerate loss in any case, making deliberate receiver misbehavior more likely and masking implementation bugs.

To combat misbehaving receivers, the data sender could verify the acknowledgements it receives and punish misbehavior. (Often it has an incentive, namely treating its other clients fairly.) To do this the sender provides an unpredictable nonce with each packet, which the receiver echoes in its acknowledgements. DCCP uses the ECN Nonce [22] for this purpose; it encodes one bit of unpredictable information that is destroyed by loss or ECN marking.

This need for verification affected many aspects of the protocol, including things as fundamental as the definition of when packets are received for the purposes of acknowledgement.

The receiver might report each acknowledged packet's ECN Nonce bit individually, but this limits the compressibility of acknowledgements. Instead, as with TCP, the receiver reports an ECN Nonce Echo, in this case the exclusive-or of the ECN Nonces on all the packets positively acknowledged by a particular acknowledgement. DCCP's acknowledgement options encode this single bit either via option number (as in "Ack Vector [Nonce Echo 0]" and "Ack Vector [Nonce Echo 1]") or by using a bit of option data.

DCCP simplifies nonce verification somewhat relative to TCP. Nonces are assigned per packet, but TCP acknowledgements refer to byte-based sequence numbers, making the identities of the acknowledged packets potentially ambiguous (if packets have overlapping sequence numbers, for example). In practice, this is not a major problem, but DCCP's per-packet sequence numbers match more cleanly with per-packet nonces.

5.1.1 When are packets received?

Protocol descriptions must define when a packet may be acknowledged. This often depends both on the network and on the end host. For example, TCP data must not be acknowledged until it is guaranteed that the application will get that data (assuming it wants the data); a TCP receive buffer is not allowed to drop data after acknowledging it.[6]

[6] This refers only to the cumulative acknowledgement: data acknowledged by SACKs may be dropped.

DCCP is unreliable, though, and one reason for this is to minimize the delivery of old data to an application. Thus, a DCCP stack might allow the application to request a front-drop receive buffer, explicitly preferring new data to old.

How should such receive-buffer-dropped packets be acknowledged? Essentially there are two separate events: delivery from the network and delivery to the application. Network congestion control cares about loss rate and round-trip time based on the former, whereas the sending application cares only about the latter. We want to tell the sending application that the receiving application did not receive the packet's payload, while simultaneously telling the sending DCCP stack that the drop was not due to congestion.

Since the receiving endpoint is claiming that the network delivered the dropped packets, it must report their ECN Nonces; to do otherwise would allow a misbehaving receiver to cheat. The cleanest way to implement this is to report valid packets as "received" (or "received ECN marked") when they have been received by the DCCP protocol, irrespective of whether they make it to the application. Note that this differs from TCP. (The sender can assume that DCCP options and other metadata have been processed on "received" packets.) A separate option called Data Dropped can report that "received" packets had their payload discarded before making it to the application—because of receive buffer overflow or payload corruption, for example. This division of acknowledgement information into two options, one targeted at congestion control and the other targeted at the sending application, is a classic example of separation of concerns.

5.1.2 Other misbehavior opportunities

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

5.2 Hijacking attacks

The most serious kind of attack, from a user's point of view, is connection hijacking, where the role of one endpoint in a connection is taken over by a third-party attacker without the other endpoint noticing. Hijacking attacks can insert bad data into a connection, causing inappropriate application actions or confusion and often, eventually, a more serious security breach.

An early decision was made not to include cryptographic mechanisms, such as public-key cryptography, in the base protocol, both because such mechanisms tend to be expensive and because IPsec provides similar functionality. This decision limits the kinds of threats DCCP can withstand alone. Without end-to-end cryptographic mechanisms, there is no in-band way to bootstrap security that is robust against eavesdroppers. An attacker that can read, insert, and remove arbitrary network packets can hijack any connection. We therefore explicitly limit the threats DCCP will address: if an attacker can snoop packets, all bets are off.

TCP addresses hijacking attacks with its sequence numbers. An attacker that can guess a connection's sequence numbers can force its way into an existing connection [15], although the partner TCP will usually notice the intrusion. Modern TCPs address this problem with randomly-chosen initial sequence numbers [4]. DCCP adopts a similar solution.

5.2.1 Sequence number validity

Unfortunately, DCCP's unreliability and per-packet sequence numbers add a wrinkle, namely determining sequence number validity: How can we tell whether a packet with sequence number s is a valid part of the current connection? It might, instead, be an old packet dating from a previous connection with the same endpoints and ports, or an attack packet.

TCP's receive window defines its range of valid sequence numbers. We note that it sometimes difficult to set the receive window correctly, so receive windows often limit data transfer rates [23], probably inappropriately.

DCCP is a congestion-control protocol, not a flow-control protocol; applications using unreliable transport would generally prefer to receive the most recent data, making TCP-style flow control inappropriate. Thus, instead of a receive window for flow control, DCCP defines a Loss Window used only to define the valid sequence number range. This will generally be set larger than TCP's receive window, as the goal is to cover a number of RTTs of packets in the window. Unlike TCP's receive window, it always moves up so that the greatest valid sequence number received is somewhere in the middle of the window.

Unfortunately, DCCP's per-packet sequence numbers mean that a DCCP endpoint can get out of sync with its partner's sequence numbers no matter how big the Loss Window is. Imagine that the link from A to B starts drop-

ping all packets. Every packet sent gets a new sequence number—even acks get sequence numbers—so A's packets will, by necessity, eventually pass B's Loss Window, even if those packets are only probes sent to test link status.

DCCP connections therefore need some mechanism for getting their endpoints back into sync after a burst of loss. A DCCP endpoint cannot wholly discard a packet with a bad sequence number. Instead, it responds to its partner with a challenge, which says, effectively, "Is your sequence number really that far off from the sequence number I expect? My sequence number is s." If the partner replies "Yes" while acknowledging s, the connection can be updated with the new sequence number. All intervening packets, including the packet that prompted the challenge, are considered lost.

This naive mechanism provides security against hijackers that cannot snoop. For instance, an attacker still needs to guess one valid sequence number to be successful: the acknowledgement number s. DCCP actually supports a slightly more complex identification option, an MD5 hash of two shared secrets (and the sequence and acknowledgement numbers, to prevent replay), but this provides only incrementally more security unless an out-of-band mechanism is used to exchange the secrets.

5.2.2 Mobility

DCCP also provides support for mobility; a mobility-capable endpoint can move to a new IP address, then inform its partner of the new address. This somewhat resembles Mobile IP's semantics. (SCTP's more complex mechanism supports, for example, multiple IP addresses for redundancy: retransmissions are sent to secondary addresses.) Mobility increases both the protocol's functionality relative to TCP and the possibilities for attack. In particular, an attacker that can guess sequence numbers might move a DCCP connection to its own address, leaving the real endpoint with no idea where the connection has gone. We partially address this by forcing DCCP-Move packets to carry valid identification options, such as the MD5 hash mentioned above; an attacker that can't generate identification options can't inappropriately move connections.

But mobility might open the door for other kinds of attacks. For example, DCCP totally ignores inappropriate DCCP-Move packets. This has a security basis: if DCCP responded (for instance, by sending resets), attackers could extract information from a connection, such as currently valid sequence numbers, simply by sending inappropriate Move packets.

5.2.3 Summary

To summarize DCCP's anti-hijacking properties:

- Connections are meant to be safe against attackers who cannot guess valid sequence numbers (for instance, by snooping).
- DCCP does not leak sequence numbers to IP addresses not already involved in the connection.
- DCCP never reveals the shared secrets used to generate identification options.

Note also the limited coverage of the MD5 hash, namely the shared secrets and the sequence and acknowledgement numbers. It might seem better to cover the whole packet, preventing data manipulations; but this would break DCCP's ability to traverse conventional network address translators, which modify packets' addresses, ports, and checksums. This illustrates the tangled world in which a modern transport protocol must live. True end-to-end security should be provided by IPsec, but under certain circumstances IPsec is not usable; and in some cases application-level authentication would provide most of the benefits. A transport protocol needs to remain as agnostic to these issues as possible while still being robust.

5.3 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.

First, half-open connections: When responding to a DCCP-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 [5], this lets the server keep no information whatsoever about half-open connections; unlike SYN cookies, it can encapsulate lots of state.

DCCP servers can also require that cooperating clients hold the Time-Wait state for a connection. (This state remains at one of the two endpoints for at least two minutes, to prevent confusion in case the network delivers packets late.) A DCCP connection is closed with the sequence Close–Reset. The receiver of Close, and sender of Reset,

need not hold the Time-Wait state, and the protocol is designed so that a server can force the client to send the Close.

Attackers might also overwhelm a DCCP endpoint by making it perform expensive calculations or actions, such as checking or generating MD5 hashes in DCCP's mobility or challenge mechanisms. The first line of defense here is the sequence number validity checks; if a connection is currently live and active, then there is no way the other endpoint can have lost sequence number validity, and so any packets with invalid sequence numbers that require checking or hash generation can be ignored. In the case where an attacker can cause a sequence number validity failure (by flooding a link for example), we require a challenge to include an MD5 identification hash. Receipt of a bad challenge may cause the receiver to ignore challenges for a short time; thus while a determined attacker can prevent the DCCP session recovering, the other sessions and tasks the victim is handling are protected.

6 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. We note that DCCP is still a work-in-progress; it may change further, but many of the fundamental issues we've discussed should be relevant for any protocol attempting to operate in this space.

Acknowledgements

DCCP has benefitted from conversations and feedback with many people, including Jitendra Padhye (our coauthor on some of the DCCP documents), Aaron Falk, and members of the DCCP Working Group, the Transport Area Working Group, and the End-to-End Research Group.

References

[1] 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.

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

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

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

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

[6] 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.

[7] S. Floyd, M. Handley, and E. Kohler. Problem statement for DCCP. Internet-Draft draft-ietf-dccp-problem-00, Internet Engineering Task Force, Oct. 2002. Work in progress.

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

[9] 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.

[10] 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.

[11] C. Huitema. The case for packet level FEC. In *Protocols for High-Speed Networks*, pages 109–120, 1996.

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

[13] 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.

[14] L.-A. Larzon, M. Degermark, S. Pink, L.-E. Jonsson, and G. Fairhurst. The UDP-Lite protocol. Internet-Draft draft-ietf-tsvwg-udp-lite-01, Internet Engineering Task Force, Dec. 2002. Work in progress.

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

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

[17] 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.

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

[19] 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.

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

[21] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. RFC 1889, Internet Engineering Task Force, Jan. 1996.

[22] N. Spring, D. Wetherall, and D. Ely. Robust ECN signaling with nonces. Internet-Draft draft-ietf-tsvwg-tcp-nonce-04.txt, Internet Engineering Task Force, Oct. 2002. Work in progress.

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