Internet Engineering Task Force
INTERNET-DRAFT
draft-ietf-dccp-spec-02.ps

Eddie Kohler
Mark Handley
Sally Floyd
ICIR
Jitendra Padhye
Microsoft Research
9 May 2003
Expires: November 2003

Datagram Congestion Control Protocol (DCCP)

Status of this Document

Abstract

This document specifies the Datagram Congestion Control Protocol (DCCP), which implements a congestion-controlled, unreliable flow of datagrams suitable for use by applications such as streaming media.

TO BE DELETED BY THE RFC EDITOR UPON PUBLICATION:

Changes since draft-ietf-dccp-spec-01.txt:

* Revise definition of when packets are reported as received, due to ECN Nonce verification problems with the previous definition and options.

* Replace Receive Buffer Drops with Data Dropped.

* Remove Data Discarded in favor of Data Dropped with Drop State 0.

* Remove Buffer Closed in favor of Data Dropped with Drop State 4.

* Add Initial Sequence Number setting guidelines.

* Add sections on retransmission of Requests, and a table to the state diagram.

* Made the 4-bit Reserved field in the DCCP generic header available for use by CCIDs.

* Refine description of CCID 1.

* Add Middlebox Considerations.

* Change Identification option to allow middleboxes to change port numbers, DCCP options, and/or packet data without disrupting the connection.

* Specify that Ignored should be sent only on packets with Acknowledgement Numbers.

* Add Aggression Penalty Reset Reason.

* Add Payload Checksum option.

* Add Elapsed Time option (formerly specific to CCID 3).

* Timestamp Echo option can omit Elapsed Time, or provide a two-byte Elapsed Time value. Elapsed Time is measured in tenths of milliseconds, not microseconds.

* Clean up DCCP-Move and feature-negotiation options discussions.

* Confirm(Connection Nonce) sends no data.

* Ack Vector implementation supports ECN Nonce Echo.

* Add CSlen and Partial Checksumming Design Motivation.

* Clarify that Ack Vectors may be sent even if Use Ack Vector is false.

Table of Contents

1. Introduction

This document specifies the Datagram Congestion Control Protocol (DCCP). DCCP provides the following features:

- An unreliable flow of datagrams, with acknowledgements.
- A reliable handshake for connection setup and teardown.
- Reliable negotiation of options, including negotiation of a suitable congestion control mechanism.
- Mechanisms allowing a server to avoid holding any state for unacknowledged connection attempts or already-finished connections.
- Optional mechanisms that tell the sender, with high reliability, which packets reached the receiver, and whether those packets were ECN marked, corrupted, or dropped in the receive buffer.
- Congestion control incorporating Explicit Congestion Notification (ECN) and the ECN Nonce, as per [RFC 3168] and [ECN NONCE].
- Path MTU discovery, as per [RFC 1191].

DCCP is intended for applications that require the flow-based semantics of TCP, but which do not want TCP's in-order delivery and reliability semantics, or which would like different congestion control dynamics than TCP. Similarly, DCCP is intended for applications that do not require features of SCTP [RFC 2960] such as sequenced delivery within multiple streams.

Applications that could make use of DCCP include those with timing constraints on the delivery of data such that reliable in-order delivery, when combined with congestion control, is likely to result in some information arriving at the receiver after it is no longer of use. Such applications might include streaming media and Internet telephony.

To date most such applications have used either TCP, with the problems described above, or used UDP and implemented their own congestion control mechanisms (or no congestion control at all). The purpose of DCCP is to provide a standard way to implement congestion control and congestion control negotiation for such applications. One of the motivations for DCCP is to enable the use of ECN, along with conformant end-to-end congestion control, for applications that would otherwise be using UDP. In addition, DCCP implements reliable connection setup, teardown, and feature negotiation.

A DCCP connection contains acknowledgement traffic as well as data traffic. Acknowledgements inform a sender whether its packets arrived, and whether they were ECN marked. Acks are transmitted as reliably as the congestion control mechanism in use requires, possibly completely reliably.

Previous drafts of this specification called the protocol DCP, or Datagram Control Protocol. The name was changed to make the acronym sound less like "TCP".

2. Design Rationale

DCCP is intended to be used by applications that currently use UDP without end-to-end congestion control. The desire is for many applications to have little reason not to use DCCP instead of UDP, once DCCP is deployed. Thus, DCCP was designed to have as little overhead as possible, in terms both of the size of the packet header and in terms of the state and CPU overhead required at the end hosts.

This desire for minimal overhead results in the design decision to include only the minimal necessary functionality in DCCP, leaving other functionality, such as FEC or semireliability, to be layered on top of DCCP as desired. The desire for minimal overhead is also one of the reasons to propose DCCP instead of just proposing an unreliable version of SCTP for applications currently using UDP.

A second motivation behind the design of DCCP is to allow applications to choose an alternative to the current TCP-style congestion control that halves the congestion window in response to a congestion indication. DCCP lets applications choose between several forms of congestion control. One choice, TCP-like congestion control, halves the congestion window in response to a packet drop or mark, as in TCP. A second alternative, TFRC (TCP-Friendly Rate Control, a form of equation-based congestion control), minimizes abrupt changes in the sending rate while maintaining longer-term fairness with TCP.

In proposing a new transport protocol, it is necessary to justify the design decision not to require the use of the Congestion Manager, as well as the design decision to add a new transport protocol to the current family of UDP, TCP, and SCTP. The Congestion Manager [RFC3124] allows multiple concurrent streams between the same sender and receiver to share congestion control. However, the current Congestion Manager can only be used by applications that have their own end-to-end feedback about packet losses, and this is not the case for many of the applications currently using UDP. In addition, the current Congestion Manager does not lend itself to the use of forms of TFRC where the state about past packet drops or marks is maintained at the receiver rather than at the sender. While DCCP should be able to make use of CM where desired by the application, we do not see any benefit in making the deployment of DCCP contingent on the deployment of CM itself.

3. Concepts and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC 2119].

3.1. Anatomy of a DCCP Connection

Each DCCP connection runs between two endpoints, which we often name DCCP A and DCCP B. Data may pass over the connection in either or both directions. The DCCP connection between DCCP A and DCCP B consists of four sets of packets, as follows:

(1) Data packets from DCCP A to DCCP B.

(2) Acknowledgements from DCCP B to DCCP A.

(3) Data packets from DCCP B to DCCP A.

(4) Acknowledgements from DCCP A to DCCP B.

We use the following terms to refer to subsets and endpoints of a DCCP connection.

Subflows
A subflow consists of either data or acknowledgement packets, sent in one direction. Each of the four sets of packets above is a subflow. (Subflows may overlap to some extent, since acknowledgements may be piggybacked on data packets.)

Sequences
A sequence consists of all packets sent in one direction, regardless of whether they are data or acknowledgements. The sets 1+4 and 2+3, above, are sequences. Each packet on a sequence has a different sequence number.

Half-connections

A half-connection consists of the data packets sent in one direction, plus the corresponding acknowledgements. The sets 1+2 and 3+4, above, are half-connections. Half-connections are named after the direction of data flow, so the A-to-B half-connection contains the data packets from A to B and the acknowledgements from B to A.

HC-Sender and HC-Receiver

In the context of a single half-connection, the HC-Sender is the endpoint sending data, while the HC-Receiver is the endpoint sending acknowledgements. For example, in the A-to-B half-connection, DCCP A is the HC-Sender and DCCP B is the HC-Receiver.

3.2. Congestion Control

Each half-connection is managed by a congestion control mechanism. The endpoints negotiate these mechanisms at connection setup; the mechanisms for the two half-connections need not be the same.

Conformant congestion control mechanisms correspond to single-byte congestion control identifiers, or CCIDs. The CCID for a half-connection describes how the HC-Sender limits data packet rates; how it maintains necessary parameters, such as congestion windows; how the HC-Receiver sends congestion feedback via acknowledgements; and how it manages the acknowledgement rate. Section 7 introduces the currently allocated CCIDs, which are defined in separate profile documents.

3.3. Connection Initiation and Termination

Every DCCP connection is actively initiated by one DCCP, which connects to a DCCP socket in the passive listening state. We refer to the active endpoint as "the client" and the passive endpoint as "the server". Most of the DCCP specification is indifferent to whether a DCCP is client or server. However, only the server may generate a DCCP-CloseReq packet. (A DCCP-CloseReq packet forces the receiving DCCP to close the connection and maintain connection state for a reasonable time, allowing old packets to clear the network.) This means that the client cannot force the server to maintain connection state after the connection is closed.

DCCP does not support TCP-style simultaneous open. In particular, a host MUST NOT respond to a DCCP-Request packet with a DCCP-Response packet unless the destination port specified in the DCCP-Request corresponds to a local socket opened for listening.

DCCP does not support half-open connections either. That is, DCCP shuts down both half-connections as a unit. However, DCCP SHOULD allow applications to declare that they are no longer interested in receiving data. This would allow DCCP implementations to streamline state for certain half-connections. See Section 8.7, on the Data Dropped option---and particularly its Drop State 4---for more information.

3.4. Features

DCCP uses a generic mechanism to negotiate connection properties, such as the CCIDs active on the two half-connections. These properties are called features. (We reserve the term "option" for a collection of bytes in some DCCP header.) A feature name, such as "CCID", generally corresponds to two features, one per half-connection. For instance, there are two CCIDs per connection. The endpoint in charge of a particular feature is called its feature location.

The Change, Prefer, and Confirm options negotiate feature values. Change is sent to a feature location, asking it to change its value for the feature. The feature location may respond with Prefer, which asks the other endpoint to Change again with different values, or it may change the feature value and acknowledge the request with Confirm. Retransmissions make feature negotiation reliable. Section 6.3 describes these options further.

4. DCCP Packets

DCCP has nine different packet types:

- DCCP-Request
- DCCP-Response
- DCCP-Data
- DCCP-Ack
- DCCP-DataAck
- DCCP-CloseReq
- DCCP-Close
- DCCP-Reset
- DCCP-Move

Only the first eight types commonly occur. The DCCP-Move packet is used to support multihoming and mobility.

The progress of a typical DCCP connection is as follows. (This description is informative, not normative.)

(1) The client sends the server a DCCP-Request packet specifying the client and server ports, the service being requested, and any features being negotiated, including the CCID that the client would like the server to use. The client may optionally piggyback some data on the DCCP-Request packet---an application-level request, say---which the server may ignore.

(2) The server sends the client a DCCP-Response packet indicating that it is willing to communicate with the client. The response indicates any features and options that the server agrees to, begins or continues other feature negotiations if desired, and optionally includes an Init Cookie that wraps up all this information and which must be returned by the client for the connection to complete.

(3) The client sends the server a DCCP-Ack packet that acknowledges the DCCP-Response packet. This acknowledges the server's initial sequence number and returns the Init Cookie if there was one in the DCCP-Response. It may also continue feature negotiation.

(4) Next comes zero or more DCCP-Ack exchanges as required to finalize feature negotiation. The client may piggyback an application-level request on its final ack, producing a DCCP-DataAck packet.

(5) The server and client then exchange DCCP-Data packets, DCCP-Ack packets acknowledging that data, and, optionally, DCCP-DataAck packets containing piggybacked data and acknowledgements. If the client has no data to send, then the server will send DCCP-Data and DCCP-DataAck packets, while the client will send DCCP-Acks exclusively.

(6) The server sends a DCCP-CloseReq packet requesting a close.

(7) The client sends a DCCP-Close packet acknowledging the close.

(8) The server sends a DCCP-Reset packet whose Reason field is set to "Closed", and clears its connection state.

(9) The client receives the DCCP-Reset packet and holds state for a reasonable interval of time to allow any remaining packets to clear the network.

An alternative connection closedown sequence is initiated by the client:

(6) The client sends a DCCP-Close packet closing the connection.

(7) The server sends a DCCP-Reset packet with Reason field set to "Closed" and clears its connection state.

(8) The client receives the DCCP-Reset packet and holds state for a reasonable interval of time to allow any remaining packets to clear the network.

This arrangement of setup and teardown handshakes permits the server to decline to hold any state until the handshake with the client has completed, and ensures that the client must hold the TimeWait state at connection closedown.

4.1. Examples of DCCP Congestion Control

Before giving the detailed specifications of DCCP, we present two more detailed examples showing DCCP congestion control in operation. Again, these examples are informative, not normative.

4.1.1. DCCP with TCP-like Congestion Control

The first example is of a connection where both half-connections use TCP-like Congestion Control, specified by CCID 2 [CCID 2 PROFILE]. In this example, the client sends an application-level request to the server, and the server responds with a stream of data packets. This example is of a connection using ECN.

(1) The client sends the DCCP-Request, which includes a Change option asking the server to use CCID 2 for the server's data packets, and a Prefer option informing the server that the client would like to use CCID 2 for the its data packets.

(2) The server sends a DCCP-Response, including a Confirm option indicating that the server agrees to use CCID 2 for its data packets, and a Change option indicating that the server agrees to the client's suggestion of CCID 2 for the client's data packets.

(3) The client responds with a DCCP-DataAck acknowledging the server's initial sequence number, and including a Confirm option finalizing the negotiation of the client-to-server CCID, and an application-level request for data. We will not discuss the client-to-server half-connection further in this example.

(4) The server sends DCCP-Data packets, where the number of packets sent is governed by a congestion window, as in TCP. The details of the congestion window are defined in the profile for CCID 2, which is a separate document [CCID 2 PROFILE]. The server also sends Ack Ratio feature options specifying the number of server data packets to be covered by an Ack packet from the client. Some of these data packets are DCCP-DataAcks acknowledging packets from the client.

Each DCCP-Data and DCCP-DataAck packet is sent as ECN-Capable, with either the ECT(0) or the ECT(1) codepoint set, as described in [ECN NONCE].

(5) The client sends a DCCP-Ack packet acknowledging the data packets for every Ack Ratio data packets transmitted by the server. Each DCCP-Ack packet uses a sequence number and contains an Ack Vector, as defined in Section 8 on Acknowledgements. These packets also include Confirm options answering any Ack Ratio requests from the server.

The client's DCCP-Acks are also sent as ECN-Capable, with either ECT(0) or ECT(1). The client's Ack Vector echoes the accumulated ECN Nonce for the server's packets.

(6) The server continues sending DCCP-Data packets as controlled by the congestion window. Upon receiving DCCP-Ack packets, the server examines the Ack Vector to learn about marked or dropped data packets, and adjusts its congestion window accordingly, as described in [CCID 2 PROFILE]. Because this is unreliable transfer, the server does not retransmit dropped packets.

(7) Because DCCP-Ack packets use sequence numbers, the server has direct information about the fraction of loss or marked DCCP-Ack packets. The server responds to lost or marked DCCP-Ack packets by modifying the Ack Ratio sent to the client, as described in [CCID 2 PROFILE]. Under certain conditions, the server must acknowledge some of the client's acknowledgements; see Section 8.1 for more information.

(8) The server estimates round-trip times and calculates a TimeOut (TO) value much as the RTO (Retransmit Timeout) is calculated in TCP. Again, the specification for this is in [CCID 2 PROFILE]. The TO is used to determine when a new DCCP-Data packet can be transmitted when the server has been limited by the congestion window and no feedback has been received from the client.

(9) The DCCP-CloseReq, DCCP-Close, and DCCP-Reset packets to close the connection are as in the example above.

4.1.2. DCCP with TFRC Congestion Control

This example is of a connection where both half-connections use TFRC Congestion Control, specified by CCID 3 [CCID 3 PROFILE].

(1) The DCCP-Request and DCCP-Response packets specifying the use of CCID 3 and the initial DCCP-DataAck packet are similar to those in the CCID 2 example above.

(2) The server sends DCCP-Data packets, where the number of packets sent is governed by an allowed transmit rate, as in TFRC. The details of the allowed transmit rate are defined in the profile for CCID 3, which is a separate document [CCID 3 PROFILE]. Each DCCP-Data packet has a sequence number and a window counter value.

Some of these data packets are DCCP-DataAck packets acknowledging packets from the client, but for simplicity we will not discuss the half-connection of data from the client to the server in this example.

The use of ECN follows TCP-like Congestion Control, above, and is described further in [CCID 3 PROFILE].

(3) The receiver sends DCCP-Ack packets at least once per round-trip time acknowledging the data packets, unless the server is sending at a rate of less than one packet per RTT, as specified by [CCID 3 PROFILE]. These acknowledgements may be piggybacked on data packets, producing DCCP-DataAck packets. Each DCCP-Ack packet uses a sequence number and identifies the most recent packet received from the server. Each DCCP-Ack packet includes feedback about the loss

event rate calculated by the client, as specified by [CCID 3 PROFILE].

(4) The server continues sending DCCP-Data packets as controlled by the allowed transmit rate. Upon receiving DCCP-Ack packets, the server updates its allowed transmit rate as specified by [CCID 3 PROFILE].

(5) The server estimates round-trip times and calculates a TimeOut (TO) value much as the RTO (Retransmit Timeout) is calculated in TCP. Again, the specification for this is in [CCID 3 PROFILE].

(6) The DCCP-CloseReq, DCCP-Close, and DCCP-Reset packets to close the connection are as in the examples above.

5. Packet Formats

5.1. Generic Packet Header

All DCCP packets begin with a generic DCCP packet header:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |           Dest Port           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Type  | CCval |              Sequence Number                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data Offset  | # NDP | Cslen |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Source and Destination Ports: 16 bits each

These fields identify the connection, similar to the corresponding fields in TCP and UDP. The Source Port represents the relevant port on the endpoint that sent this packet, the Destination Port the relevant port on the other endpoint.

Type: 4 bits

The type field specifies the type of the DCCP message. The following values are defined:

0 DCCP-Request packet.

1 DCCP-Response packet.

2 DCCP-Data packet.

3 DCCP-Ack packet.

4 DCCP-DataAck packet.

5 DCCP-CloseReq packet.

6 DCCP-Close packet.

7 DCCP-Reset packet.

8 DCCP-Move packet.

CCval: 4 bits

This field is reserved for use by the sending CCID. In particular, the A-to-B CCID's sender, which is active at DCCP A, MAY send information to the receiver at DCCP B by encoding that information in CCval. DCCP proper MUST ignore the field. If the relevant CCID does not specify its value, it SHOULD be set to zero.

Sequence Number: 24 bits

The sequence number field is initialized by a DCCP-Request or DCCP-Response packet, and increases by one (modulo 16777216) with every packet sent. The receiver uses this information to determine whether packet losses have occurred. Even packets containing no data update the sequence number. Sequence numbers also provide some protection against old and malicious packets; see Section 5.2 on sequence number validity.

Very-high-rate DCCPs may need protection against wrapped sequence numbers. For example, a 10 Gb/s flow of 1500-byte DCCP packets will send 2^24 packets in about 20 seconds. This is a long time, in terms of likely round-trip times that could possibly achieve such a sustained rate, but it is not without risk. Despite this, we leave the design of mechanisms to protect against wrapped sequence numbers for future work. In particular, if it is decided that very large packet sizes are better than very large congestion windows for very-high-bandwidth flows, then 24 bits may be enough.

The two subflows' initial sequence numbers are set by the first DCCP-Request and DCCP-Response packets sent, and SHOULD be chosen as for TCP. In particular, initial sequence number choice MUST include a random or pseudorandom component to make it harder for attackers to complete sequence number attacks [RFC 1948]. The initial sequence number chosen for a given connection identifier (source address and port plus destination address and port) SHOULD increase over time, as TCP suggests [RFC 793], to prevent inappropriate delivery of old packets.

Data Offset: 8 bits

The offset from the start of the DCCP header to the beginning of the packet's payload, measured in 32-bit words.

Number of Non-Data Packets (# NDP): 4 bits

DCCP sets this field to the number of non-data packets it has sent so far on its sequence, modulo 16. A non-data packet is simply any packet not containing user data; DCCP-Ack, DCCP-Close, DCCP-CloseReq, and DCCP-Reset are always non-data packets, while DCCP-Request, DCCP-Response, and DCCP-Move might or might not be. When sending a non-data packet, DCCP increments the # NDP counter before storing its value in the packet header.

This field can help the receiving DCCP decide whether a lost packet contained any user data. (An application may want to know when it has lost data. DCCP could report every packet loss as a potential data loss, but that would cause false loss reports when non-data packets were lost.) For example, say that packet 10 had # NDP set to 5; packet 11 was lost; and packet 12 had # NDP set to 5. Then the receiving DCCP could deduce that packet 11 contained data, since # NDP did not change. Likewise, if # NDP had gone up to 6 (and packet 12 contained user data), then packet 11 must not have contained any data.

Checksum Length (Cslen): 4 bits

The checksum length field specifies what parts of the packet are covered by the checksum field. The checksum always covers at least the DCCP header, DCCP options, and a pseudoheader taken from the network-layer header (described under Checksum below). If the checksum length field is zero, that is all the checksum covers. If the field is 15, the checksum covers the packet's payload as well, possibly with 8 bits of zero padding on the right to pad the payload to an even number of bytes. Values between 1 and 14, inclusive, indicate that the checksum additionally covers that number of initial 32-bit words of the packet's payload, padded on the right with zeros as necessary.

Values other than 15 specify that corruption is acceptable in some or all of the DCCP packet's payload. In fact, DCCP cannot even detect corruption there, unless the Payload Checksum option is used (Section 8.8). The meaning of values other than 0 and 15 should be considered experimental.

Section 18.1 further discusses the motivation of, and issues related to, partial checksums. The checksum length field was inspired by UDP-Lite [UDP-LITE].

Checksum: 16 bits

DCCP uses the TCP/IP checksum algorithm. The checksum field equals the 16 bit one's complement of the one's complement sum of all 16 bit words in the DCCP header, DCCP options, a pseudoheader taken from the network-layer header, and, depending on the value of the checksum length field, some or all of the payload. When calculating the checksum, the checksum field itself is treated as 0. If a packet contains an odd number of header and text bytes to be checksummed, 8 zero bits are added on the right to form a 16 bit word for checksum purposes. The pad byte is not transmitted as part of the packet.

The pseudoheader is calculated as for TCP. For IPv4, it is 96 bits long, and consists of the IPv4 source and destination addresses, the IP protocol number for DCCP (padded on the left with 8 zero bits), and the DCCP length as a 16-bit quantity (the length of the DCCP header with options, plus the length of any data); see Section 3.1 of [RFC 793]. For IPv6, it is 320 bits long, and consists of the IPv6 source and destination addresses, the DCCP length as a 32-bit quantity, and the IP protocol number for DCCP (padded on the left with 24 zero bits); see Section 8.1 of [RFC 2460].

Packets with invalid checksums MUST be ignored. In particular, their options MUST NOT be processed.

5.2. Sequence Number Validity

DCCP endpoints SHOULD ignore packets with invalid sequence numbers, which may arise if the network delivers a very old packet or an attacker attempts to hijack a connection. TCP solves this problem with its window. In DCCP, however, sequence numbers change with each packet sent, even pure acknowledgements. Thus, a loss event that dropped many consecutive packets could cause two DCCPs to get out of sync relative to any window.

DCCP uses Loss Window and Identification mechanisms to determine whether a given packet's sequence number is valid. Each HC-Sender gives the corresponding HC-Receiver a loss window width W; see Section 6.9. This reflects how many packets the sender expects to be in flight. Only the sender can anticipate this number. One good guideline is to set it to about 3 or 4 times the maximum number of packets the sender expects to send in any round-trip time. Too-small values increase the risk of the endpoints getting out sync after bursts of loss; too-large values increase the risk of connection hijacking. W defaults to 1000. The Identification mechanism is used to get back into sync when more than W consecutive packets are lost.

The HC-Receiver sets up a loss window of W consecutive sequence numbers containing GSN, the Greatest Sequence Number it has received on any valid packet from the sender. ("Consecutive" and "greatest" are measured in circular sequence space. The receiver may center the loss window on GSN, or arrange it asymmetrically.) Sequence numbers outside this loss window are invalid. Packets with invalid sequence numbers are themselves invalid, unless both of the following conditions are true:

(1) No valid packet has been received recently (for instance, within at least one round-trip time).

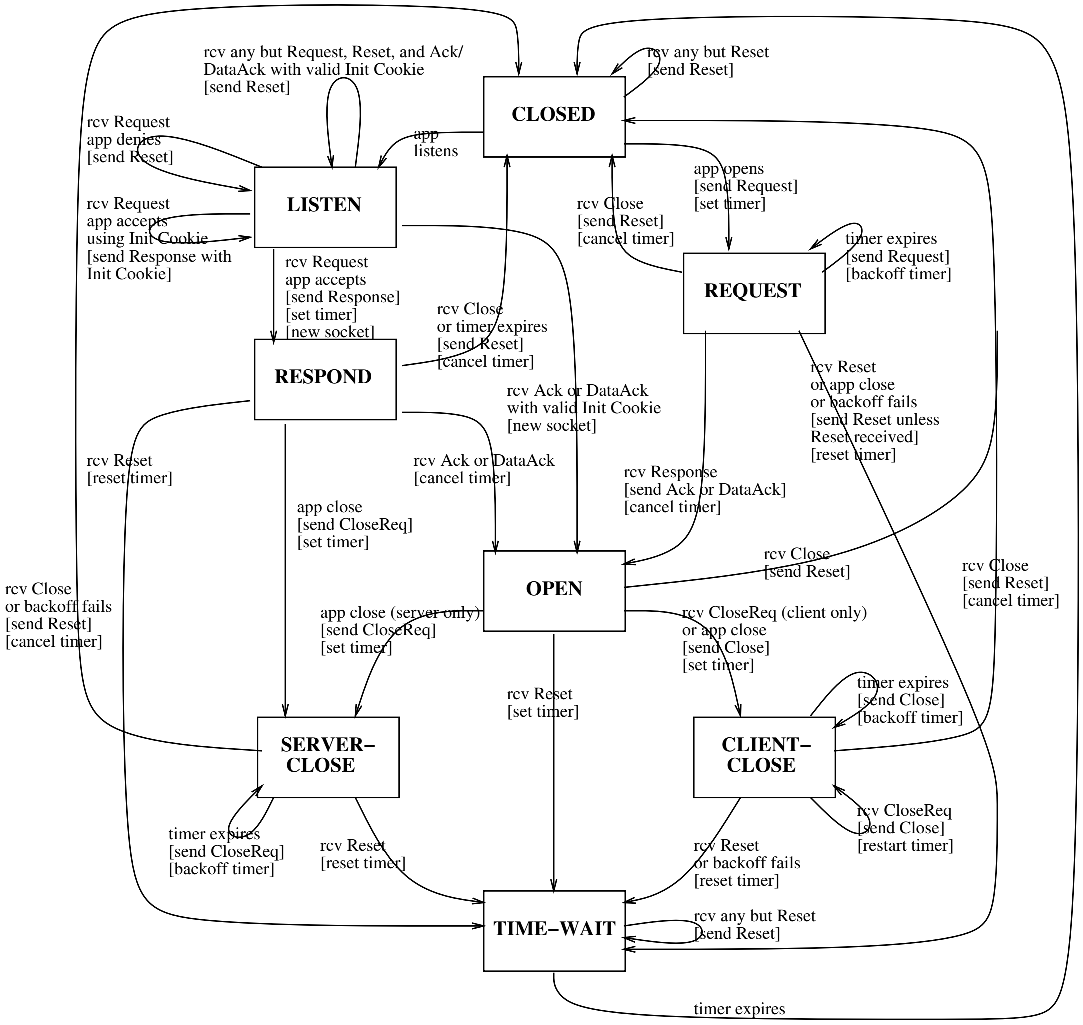
(2) The packet includes a correct Identification or Challenge option (see Section 6.4.3).

The receiving DCCP SHOULD ignore invalid packets. In particular, it SHOULD NOT pass any enclosed data to the application, update its congestion control or feature state, or close the connection. However, the receiving DCCP MAY send a DCCP-Ack packet to the sender, as allowed by the congestion control mechanism in use. This packet SHOULD contain the last received valid sequence number and a Challenge option (Section 6.4.4). The other DCCP will send an Identification option to resync.

A DCCP endpoint MAY implement rate limits to reduce the likelihood of denial-of-service attack. In particular, it MAY ignore all packets with bad sequence numbers---even those containing Identification or Challenge options---for some amount of time, on the order of one round-trip time, after receiving a packet with an invalid Identification or Challenge option; and it MAY rate-limit the Challenge options it sends.

5.3. DCCP State Diagram

In this section we present a DCCP state diagram showing how a DCCP connection should progress, and the proper responses for packets or timeout events in various connection states. The state diagram is illustrative; the text should be considered definitive.



All receive events on the diagram represent receipt of valid packets. For example, receiving a Reset with a bad Acknowledgement Number SHOULD NOT cause DCCP to transition to the Time-Wait state. DCCP implementations MAY send Acks as described above, or "Invalid Packet" Resets, in response to invalid packets; any such responses SHOULD be rate-limited.

Otherwise-valid packets without explicit transitions in the state diagram SHOULD be treated according to the table below. Particular actions are "OK", meaning the packet MUST be processed according to this document; "Rst", meaning the receiver SHOULD either ignore the packet or respond with a (rate-limited) Reset; and "-", meaning the packet SHOULD be ignored. Entries may take the form "Old/New", where "Old" applies to old packets and "New" to new packets (whose sequence numbers are greater than the largest sequence number seen so far). The table respecifies some transitions listed in the state diagram---for instance, those for receiving packets in the TIME-WAIT state. In these cases, prefer the action listed in the diagram. For example, in the TIME-WAIT case, prefer sending rate-limited Resets when valid packets are received; the table would allow ignoring them. However, either action would be acceptable.

State	Request	Response	Data/Ack/ DataAck/ Move	CloseReq	Close	Reset
CLOSED	Rst	Rst	Rst	Rst	Rst	OK
LISTEN	OK	Rst	Rst(1)	Rst	Rst	OK
REQUEST	Rst	OK	Rst	Rst	Rst	OK
RESPOND	-/OK	Rst	Rst/OK	Rst	OK	OK
OPEN (server)	-/Rst	Rst	OK	Rst	OK	OK
OPEN (client)	Rst	-/Rst	OK	OK	OK	OK
SERVER-CLOSE	-/Rst	Rst	OK	Rst	OK	OK
CLIENT-CLOSE	Rst	-/Rst	OK	OK	OK	OK
TIME-WAIT	Rst	Rst	Rst	Rst	Rst	OK

Notes: (1) Data/Ack/DataAck with valid Init Cookie OK.

The Open state does not signify that a DCCP connection is ready for data transfer. In particular, incomplete feature negotiations might prevent data transfer. Feature negotiation takes place in parallel with the state transitions on this diagram.

Only the server may take the transition from the OPEN state to the SERVER-CLOSE state. (The server is the DCCP endpoint that began in the LISTEN state.) Similarly, only the client must transition to CLIENT-CLOSE after receiving a CloseReq packet.

5.4. DCCP-Request Packet Format

A DCCP connection is initiated by sending a DCCP-Request packet. The format of a DCCP request packet is:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                 Generic DCCP Header (12 bytes)                /
/                   with Type=0 (DCCP-Request)                  /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Service Name                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Options                   /   [padding]   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              data                             |
|                               ...                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Service Name: 32 bits

The Service Name field describes the service to which the sender is trying to connect. Service Names are 32-bit numbers allocated by IANA; they are meant to correspond to application services and protocols, such as FTP and HTTP, and are not intended to be DCCP-specific. With Service Names, stateful middleboxes, such as firewalls, can identify the application running on a nonstandard port (assuming the DCCP header has not been encrypted). A Service Name of zero is a wildcard, matching any service. The host operating system MAY force every DCCP socket, both actively and passively opened, to specify a nonzero Service Name. Connection requests MUST fail if the Destination Port on the receiver has a different Service Name from that given in the packet, and both Service Names are nonzero. In this case, the receiver will respond with a DCCP-Reset packet (with Reason set to "Bad Service Name"). A server or stateful middlebox MAY also send a "Bad Service Name" DCCP-Reset in response to packets with Service Name value 0.

Options

DCCP-Request packets will usually include a "Change(Connection Nonce)" option, to inform the server of the client's connection nonce; see Section 6.4.

The client MAY send new DCCP-Request packets if no response is received after some timeout. Each retransmission MUST increment the Sequence Number, and possibly # NDP, by one. The retransmission strategy SHOULD be similar to that for retransmitting TCP SYNs.

A client MAY decide to give up after some number of DCCP-Requests. If so, it MAY send a DCCP-Reset packet to the server, to clean up state in case one or more of the Requests actually arrived. The DCCP-Reset SHOULD have Reason set to "Closed".

5.5. DCCP-Response Packet Format

In the second phase of the three-way handshake, the server sends a DCCP-Response message to the client. In this phase, a server will often specify the options it would like to use, either from among those the client requested, or in addition to those. Among these options is the congestion control mechanism the server expects to use.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                 Generic DCCP Header (12 bytes)                /
/                  with Type=1 (DCCP-Response)                  /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Reserved    |            Acknowledgement Number             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                   /   [padding]    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
|                              ...                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Acknowledgement Number: 24 bits

The Acknowledgement Number field, which appears in several packet types, acknowledges the *greatest* valid sequence number received so far on this connection. ("Greatest" is, of course, measured in circular sequence space.) In the case of a DCCP-Response packet, the acknowledgement number field will equal the sequence number from the DCCP-Request. Acknowledgement numbers make no attempt to provide precise information about which packets have arrived; options such as the Ack Vector do this.

Some care is required in defining when a packet is "received" for purposes of acknowledgement. All valid packets received by a DCCP stack MUST be acknowledged as "received", even if their payloads were dropped (due to receive buffer overflow or payload corruption, for example). The receiving DCCP MUST have processed the options on every packet it reports as "received". The Data Dropped option (Section 8.7) helps the sending application determine when packet payloads were dropped by the receiving DCCP. This issue is discussed in somewhat more detail in Section 8.5.

Reserved: 8 bits

The version of DCCP specified here SHOULD set this field to all zeroes on generated packets, and ignore its value on received packets.

Options

The Data Dropped and Init Cookie options are particularly useful for DCCP-Response packets (Sections 8.7 and 6.5). In addition, DCCP-Response, or early DCCP-Data or DCCP-Ack packets, will often include "Confirm(Connection Nonce)" and "Change(Connection Nonce)" options, to further negotiate connection nonces (Section 6.4), as well as options to negotiate CCIDs and other relevant features.

The receiver MAY respond to a DCCP-Request packet with a DCCP-Reset packet to refuse the connection. Relevant Reset Reasons for refusing a connection include "Connection Refused", when the DCCP-Request's Destination Port did not correspond to a DCCP port open for listening; "Bad Service Name", when the DCCP-Request's Service Name did not correspond to the service name registered with the Destination Port; and "Too Busy", when the server is currently too busy to respond to requests. The server SHOULD limit the rate at which it generates these resets.

The receiver SHOULD NOT retransmit DCCP-Response packets; the sender will retransmit the DCCP-Request if necessary. The responder will detect that the retransmitted DCCP-Request applies to an existing connection because of its Source and Destination Ports. Every valid DCCP-Request received MUST elicit a new DCCP-Response, unless the responder can guarantee that the requestor has received at least one Response already. (For instance, if the responder has received a valid DCCP-Data or DCCP-Ack packet from the requestor, then it knows the newly received Request is old, and SHOULD be ignored.) Each new DCCP-Response MUST increment the Sequence Number, and possibly # NDP, by one.

5.6. DCCP-Data, DCCP-Ack, and DCCP-DataAck Packet Formats

The payload of a DCCP connection is sent in DCCP-Data and DCCP-DataAck packets, while DCCP-Ack packets are used for acknowledgements when there is no payload to be sent. DCCP-Data packets look like this:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                 Generic DCCP Header (12 bytes)                /
/                    with Type=2 (DCCP-Data)                    /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Options                   /   [padding]   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
|                              ...                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

DCCP-Ack packets dispense with the data, but contain an acknowledgement number:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                 Generic DCCP Header (12 bytes)                /
/                    with Type=3 (DCCP-Ack)                     /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Reserved    |           Acknowledgement Number              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Options                   /   [padding]   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

DCCP-DataAck packets contain both data and an acknowledgement number: acknowledgement information is piggybacked on a data packet.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                 Generic DCCP Header (12 bytes)                /
/                  with Type=4 (DCCP-DataAck)                   /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Reserved    |           Acknowledgement Number              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Options                   /   [padding]   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
|                             ...                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

DCCP-Ack and DCCP-DataAck packets often include additional acknowledgement options, such as Ack Vector, as required by the congestion control mechanism in use.

DCCP A sends DCCP-Data and DCCP-DataAck packets to DCCP B due to application events on host A. These packets are congestion-controlled by the CCID for the A-to-B half-connection. In contrast, DCCP-Ack packets sent by DCCP A are controlled by the CCID for the B-to-A half-connection. Generally, DCCP A will piggyback acknowledgement information on data packets when acceptable, creating DCCP-DataAck packets. DCCP-Ack packets are used when there is no data to send from DCCP A to DCCP B, or when the link from A to B is so congested that sending data would be inappropriate.

Section 8, below, describes acknowledgements in DCCP.

A DCCP-Data or DCCP-DataAck packet may contain no data bytes if the application sends a zero-length datagram. Such zero-length datagrams MUST be reported to the receiving application.

5.7. DCCP-CloseReq and DCCP-Close Packet Format

The DCCP-CloseReq and DCCP-Close packets have the same format. However, only the server can send a DCCP-CloseReq packet. Either client or server may send a DCCP-Close packet. The receiver of a valid DCCP-Close packet SHOULD respond with a DCCP-Reset packet, with Reason set to "Closed"; the endpoint that originally sent the DCCP-Close will hold TimeWait state. The receiver of a valid DCCP-CloseReq packet SHOULD respond with a DCCP-Close packet; that receiving endpoint will expect to hold TimeWait state after later receiving a DCCP-Reset. See the state diagram in 5.3 for more information.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                 Generic DCCP Header (12 bytes)                /
/           with Type=5 or 6 (DCCP-Close or CloseReq)           /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Reserved    |            Acknowledgement Number             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Options                 /   [padding]     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

5.8. DCCP-Reset Packet Format

DCCP-Reset packets unconditionally shut down a connection. Every connection shutdown sequence ends with a DCCP-Reset, but resets may be sent for other reasons, including bad port numbers, bad option behavior, incorrect ECN Nonce Echoes, and so forth. The reason for a reset is represented by an eight-bit number, the Reason field, and 24 bits of additional data.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                 Generic DCCP Header (12 bytes)                /
/                   with Type=7 (DCCP-Reset)                    /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Reserved    |            Acknowledgement Number             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Reason     |    Data 1     |    Data 2     |    Data 3     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Options                 /   [padding]     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Reason: 8 bits
The Reason field represents the reason that the sender reset the DCCP connection.

Data 1, Data 2, and Data 3: 8 bits each
The Data fields provide additional information about why the sender reset the DCCP connection. The meanings of these fields depend on the value of Reason.

The following Reasons are currently defined. The "Data" columns describe what the Data fields should contain for a given Reason. In those columns, N/A means the Data field SHOULD be set to 0 by the sender of the DCCP-Reset, and ignored by its receiver.

Reason	Name	Data 1	Data 2	Data 3	Section Reference
0	Unspecified	N/A	N/A	N/A	
1	Closed	N/A	N/A	N/A	4
2	Invalid Packet	packet type	N/A	N/A	5.3
3	Option Error	option number	option data (if any)		
4	Feature Error	feature number	feature data (if any)		
5	Connection Refused	N/A	N/A	N/A	5.5
6	Bad Service Name	N/A	N/A	N/A	5.4
7	Too Busy	N/A	N/A	N/A	5.5
8	Bad Init Cookie	N/A	N/A	N/A	6.5
9	Invalid Move	N/A	N/A	N/A	5.9
10	Unanswered Challenge	N/A	N/A	N/A	6.4.4
11	Fruitless Negotiation	feature number	feature data (optional)		6.3.7
12	Aggression Penalty	N/A	N/A	N/A	9.2

5.9. DCCP-Move Packet Format

The DCCP-Move packet type is part of DCCP's support for multihoming and mobility, which is described further in Section 10. DCCP A sends a DCCP-Move packet to DCCP B after changing its address and/or port number. The DCCP-Move packet requests that DCCP B start sending packets to the new address and port number. The old address and port are stored explicitly in the DCCP-Move header; the new address and port come from the packet's network header and generic DCCP header. The old address's type is indicated explicitly by an Old Address Family field. The Sequence Number and Acknowledgement Number fields and a mandatory Identification option provide some protection against hijacked connections. See Section 10 for more on security and DCCP's mobility support.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                 Generic DCCP Header (12 bytes)                /
/                    with Type=8 (DCCP-Move)                    /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Reserved    |            Acknowledgement Number             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Old Address Family       |           Old Port            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                          Old Address                          /
/                                             /   [padding]     /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Options, including Identification      /   [padding]   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
|                              ...                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Old Address Family: 16 bits

The Old Address Family field indicates the address family formerly used for this connection, and takes values from the Address Family Numbers registry administered by IANA. Particular values include 1 for IPv4 and 2 for IPv6. An endpoint MUST discard DCCP-Move packets with unrecognized Old Address Family values.

Old Port: 16 bits

The former port number used by DCCP A's endpoint.

Old Address: at least 32 bits

The former address used by DCCP A's endpoint, padded on the right to a multiple of 32 bits. The form and size of the address are determined by the Old Address Family field. For instance, if Old Address Family is 1, then Old Address contains an IPv4 address and takes 32 bits; if it is 2, then Old Address contains an IPv6 address and takes 128 bits.

Options

Every DCCP-Move packet MUST include a valid Identification option (see Section 6.4).

DCCP B SHOULD respond to the DCCP-Move with a DCCP-Reset (with Reason set to "Invalid Move") if any of the following conditions holds:

(1) Neither the Old Address/Old Port combination nor the network address/Source Port combination refers to a currently active DCCP connection.

(2) The Identification option is not present or invalid.

(3) DCCP B does not support mobility, or its Mobility Capable feature is off.

After receiving such an invalid DCCP-Move, DCCP B MAY ignore subsequent DCCP-Move packets, valid or not, for a short period of time, such as one round-trip time. This protects DCCP B against denial-of-service attacks from floods of invalid DCCP-Moves.

DCCP B SHOULD respond to a valid DCCP-Move packet with a DCCP-Ack or DCCP-DataAck packet acknowledging the move. If DCCP B accepts the move, it MUST send this acknowledgement to the network address/Source Port combination; if it rejects the move, which it MAY do for any reason, it MUST send the acknowledgement to the Old Address/Old Port combination.

If the acknowledgement is lost, DCCP A might resend the DCCP-Move packet (using a new sequence number). DCCP B will detect this case because the network address/Source Port combination corresponds to a valid connection, for which the Sequence Number and Acknowledgement Number fields are valid; the Identification option is valid for that connection; and the Old Address/Old Port combination no longer refers to a valid DCCP connection. It SHOULD respond by sending another acknowledgement, as allowed by the congestion control mechanism in use.

We note that DCCP mobility, as provided by DCCP-Move, may not be useful in the context of IPv6, with its mandatory support for Mobile IP.

6. Options and Features

All DCCP packets may contain options, which occupy space at the end of the DCCP header and are a multiple of 8 bits in length. All options are always included in the checksum. An option may begin on any byte boundary.

The first byte of an option is the option type. Options with types 0 through 31 are single-byte options. Other options are followed by a byte indicating the option's length. This length value includes the two bytes of option-type and option-length as well as any option-data bytes, and MUST therefore be greater than or equal to two.

The following options are currently defined:

Type	Option Length	Meaning	Section Reference
0	1	Padding	6.1
2	1	Slow Receiver	8.6
32	3-4	Ignored	6.2
33	variable	Change	6.3
34	variable	Prefer	6.3
35	variable	Confirm	6.3
36	variable	Init Cookie	6.5
37	variable	Ack Vector [Nonce 0]	8.5
38	variable	Ack Vector [Nonce 1]	8.5
39	variable	Data Dropped	8.7
40	6	Timestamp	6.6
41	6-10	Timestamp Echo	6.8
42	variable	Identification	6.4.3
44	variable	Challenge	6.4.4
45	4	Payload Checksum	8.8
46	4-6	Elapsed Time	6.7
128-255	variable	CCID-specific options	7.4

6.1. Padding Option

The padding option, with type 0, is a single byte option used to pad between or after options. It either ensures the payload begins on a 32-bit boundary (as required), or ensures alignment of following options (not mandatory).

```
+--------+
|00000000|
+--------+
  Type=0
```

6.2. Ignored Option

The Ignored option, with type 32, signals that a DCCP did not understand some option. This can happen, for example, when a conventional DCCP converses with an extended DCCP. Each Ignored option has one or two bytes of data. The first byte contains the offending option type; the second, if present, contains the first byte of the offending option's data. If the offending option had no data, the Ignored option MAY still supply two bytes of data, with the second byte set to 0.

Ignored options SHOULD be sent only on packets that contain Acknowledgement Numbers (that is, DCCP-Reponse, DCCP-Ack, DCCP-DataAck, DCCP-Close, DCCP-CloseReq, DCCP-Reset, and DCCP-Move), and SHOULD concern options sent on the packet acknowledged by the Acknowledgement Number.

```
+--------+--------+--------+
|00100000|00000011|Opt Type|
+--------+--------+--------+
 Type=32  Length=3

+--------+--------+--------+--------+
|00100000|00000100|Opt Type|Opt Data|
+--------+--------+--------+--------+
 Type=32  Length=4
```

6.3. Feature Negotiation

DCCP contains a mechanism for reliably negotiating features, notably the congestion control mechanism in use on each half-connection. The motivation is to implement reliable feature negotiation once, so that different options need not reinvent that wheel.

Three options, Change, Prefer, and Confirm, implement feature negotiation. Change is sent to a feature's location, asking it to change the feature's value. The feature location may respond with Prefer, which asks the other endpoint to Change again with different values, or it may change the feature value and acknowledge the request with Confirm.

Feature values MUST NOT change apart from feature negotiation, and enforced retransmissions make feature negotiation reliable. This ensures that both endpoints eventually agree on every feature's value.

Some features are non-negotiable, meaning that the feature location MUST set its value to whatever the other endpoint requests. For non-negotiable features, the feature location MUST respond to Change options with Confirm; Prefer is not useful. These features use the feature framework simply to achieve reliability.

Negotiations for multiple features may take place simultaneously. For instance, a packet may contain multiple Change options that refer to different features.

Feature negotiation generally takes place using packet types that carry no user data, such as DCCP-Ack, particularly when the relevant feature may affect how data will be treated.

6.3.1. Feature Numbers

The first data byte of every Change, Prefer, or Confirm option is a feature number, defining the type of feature being negotiated. The remainder of the data gives one or more values for the feature, and is interpreted according to the feature. The current set of feature numbers is as follows:

Number	Meaning	Neg.?	Section Reference
1	Congestion Control (CC)	Y	7
2	ECN Capable	Y	9.1
3	Ack Ratio	N	8.3
4	Use Ack Vector	Y	8.4
5	Mobility Capable	Y	10.1
6	Loss Window	N	6.9
7	Connection Nonce	N	6.4.2
8	Identification Regime	Y	6.4.1
128-255	CCID-Specific Features	?	7.4

The "Neg[otiable]?" column is "Y" for normal features and "N" for non-negotiable features.

6.3.2. Change Option

DCCP A sends a Change option to DCCP B to ask it to change the value of some feature located at DCCP B. DCCP B SHOULD respond to a Change option for a known feature with either Prefer or Confirm. In special circumstances, such as a Change option whose value is inappropriate for the listed feature number or a negotiation that seems to be going on forever, DCCP B MAY respond instead by ignoring the Change (with or without sending an Ignored option), or by resetting the connection with Reason set to "Fruitless Negotiation" or "Feature Error". DCCP A SHOULD retransmit the Change option until it receives some relevant response. DCCP A will always generate a Change option in response to a Prefer option; it may also generate a Change option due to some application event.

```
+--------+--------+--------+--------+--------+--------
|00100001| Length |Feature#| Value or Values ...
+--------+--------+--------+--------+--------+--------
 Type=33
```

6.3.3. Prefer Option

DCCP A sends a Prefer option to DCCP B to ask it to choose another value for some feature located at DCCP B. DCCP B SHOULD respond to a valid Prefer option with a Change; other possible responses include ignoring the option, sending an Ignored option, or resetting the connection, as described above. DCCP A SHOULD retransmit the Prefer option until it receives some relevant response. DCCP A may generate a Prefer option in response to some Change option, or in response to some application event. Prefer options are not useful for non-negotiable features.

```
+--------+--------+--------+--------+--------+--------
|00100010| Length |Feature#| Value or Values ...
+--------+--------+--------+--------+--------+--------
 Type=34
```

6.3.4. Confirm Option

DCCP A sends a Confirm option to DCCP B to inform it that a Change option for some feature located at DCCP A has been accepted. Generally the Confirm option will include the feature's accepted value. For some special features, such as Connection Nonce, a Confirm option contains no data; these features are identified explicitly. DCCP A MUST generate Confirm options only in response to valid Change options. DCCP A SHOULD NOT retransmit Confirm options: DCCP B will retransmit the relevant Changes as necessary. The receipt of a valid Confirm option ends the negotiation over a feature's value.

```
+--------+--------+--------+--------+--------+--------
|00100011| Length |Feature#| Value ...
+--------+--------+--------+--------+--------+--------
 Type=35
```

6.3.5. Example Negotiations

This section demonstrates several negotiations of the congestion control feature for the A-to-B half-connection. (This feature is located at DCCP A.) In this sequence of packets, DCCP A is happy with DCCP B's suggestion of CC mechanism 2:

```
B > A    Change(CC, 2)
A > B    Confirm(CC, 2)
```

Here, A and B jointly settle on CC mechanism 5:

```
B > A    Change(CC, 3, 4)
A > B    Prefer(CC, 1, 2, 5)
B > A    Change(CC, 5)
A > B    Confirm(CC, 5)
```

In this sequence, A refuses to use CC mechanism 5. If this sequence continued, one or the other endpoint would eventually abort the connection via a DCCP-Reset packet with Reason set to "Fruitless Negotiation":

```
B > A    Change(CC, 3, 4, 5)
A > B    Prefer(CC, 1, 2)
B > A    Change(CC, 5)
A > B    Prefer(CC, 1, 2)
```

Here, A elicits agreement from B that it is satisfied with congestion control mechanism 2:

```
A > B    Prefer(CC, 1, 2)
B > A    Change(CC, 2)
A > B    Confirm(CC, 2)
```

6.3.6. Unknown Features

If a DCCP receives a Change or Prefer option referring to a feature number it does not understand, it MUST respond with an Ignored option. This informs the remote DCCP that the local DCCP does not implement the feature. No other action need be taken. (Ignored may also indicate that the DCCP endpoint could not respond to a CCID-specific feature request because the CCID was in flux; see Section 7.4.)

6.3.7. State Diagram

These state diagrams present the legal transitions in a DCCP feature negotiation. They define DCCP's states and transitions with respect to the negotiation of a single feature it understands. There are two diagrams, corresponding to the two endpoints: the feature location DCCP A, and what we call the "feature requester", DCCP B.

Transitions between states are triggered by receiving a packet ("`RECV`") or by an application event ("`APP`"). Received packets are further distinguished by any options relevant to the feature being negotiated. "`RECV -`" means the packet contained no relevant option. "`RECV Chg`" denotes a Change option, "`RECV Pr`" a Prefer option, and "`RECV Cfm`" a Confirm option. The data contained in an option is given in parentheses when necessary. The "`SEND`" action indicates which option the DCCP will send next. Finally, the "`SET-VALUE`" action causes the DCCP to change its value for the relevant feature.

"`SEND`" does not force DCCP to immediately generate a packet; rather, it says which feature option must be sent on the next packet generated. A DCCP MAY choose to generate a packet in response to some "`SEND`" action. However, it MUST NOT generate a packet if doing so would violate the congestion control mechanism in use.

The requester, DCCP B, has four states: Known, Unknown, Failed, and Changing. Similarly, the feature location, DCCP A, has four states: Known, Unknown, Failed, and Confirming. In both cases, Known denotes a state where the DCCP knows the feature's current value, and believes that the other DCCP agrees. Changing and Confirming denote states where the DCCPs are in the process of negotiating a new value for the feature. The Unknown state can occur only at connection setup time. It denotes a state where the DCCP does not know any value for the feature, and has not yet entered a negotiation to determine its value. Finally, the Failed state represents a state where the other DCCP does not implement the feature under negotiation.

A DCCP may start in either the Unknown or Known state, depending on the feature in question. In particular, some features have a well-known value for new connections, in which case the DCCPs begin the connection in the Known states.

```
                    REQUESTER STATE DIAGRAM (DCCP B)

                        +-----------+
                        |  Unknown  |
                        +-----------+
      +----------+            |                    +-----------+
      |          |RECV -      |RECV -/Pr | APP     |           |RECV Pr/Cfm
      V          |SEND -      |SEND Chg            V           |SEND Chg
+-----------+    |            |             +------------+     |
|           |----+            +------------>|            |-----+
|   Known   |------------------------------>|  Changing  |
|           |        RECV Pr | APP          |            |-----+
+-----------+          SEND Chg             +------------+     |RECV -
      ^                                          | | ^         |SEND -/Chg
      |                                          | | |         |
      +------------------------------------------+ | +---------+
                       RECV Cfm(O)                 |          +----------+
                       SEND -                      +--------->|  Failed  |
                       SET-VALUE O                  RECV Ign  +----------+
                                                    SEND -


                  FEATURE LOCATION STATE DIAGRAM (DCCP A)
(O represents any feature value acceptable to DCCP A; X is not acceptable.)


        RECV Chg(O)
        SEND Cfm(O)                   RECV -  |  APP
        SET-VALUE O     +-----------+ SEND Pr(O)
   +--------------------|  Unknown  |------------+
   |                    +-----------+            |
   |     +-------+            |                  | +-----------+
   |     |       |RECV -      |RECV Chg(X)       | |           |RECV Chg(X)
   V     V       |SEND -      |SEND Pr(O)        V V           |SEND Pr(O)
+-----------+    |            |             +------------+     |  (need not be
|           |----+            +------------>|            |-----+   the same O)
|   Known   |------------------------------>| Confirming |
|           |----+     RECV Chg  |  APP     |            |-----+
+-----------+    |        SEND Pr(O)        +------------+     |RECV -
   ^     ^       |                               | | ^         |SEND -/Pr(O)
   |     |       |RECV Chg(O)                    | | |         |
   |     |       |SEND Cfm(O)                    | | +---------+
   |     |       |SET-VALUE O                    | |
   |     +-------+                               | |         +----------+
   +---------------------------------------------+ +-------->|  Failed  |
                  RECV Chg(O)                      RECV Ign +----------+
                  SEND Cfm(O)                      SEND -
                  SET-VALUE O
```

This specification allows several choices of action in certain states. The implementation will generally use feature-specific information to decide how to respond. For example, DCCP A in the Known state may respond to a Change option with either Confirm or Prefer. If DCCP A is willing to set the feature to the value specified by Change, it will generally send Confirm; but if it would like to negotiate further, it will send Prefer.

DCCP B retransmits Change options, and DCCP A retransmits Prefer options, until receiving a relevant response. However, they need not retransmit the option on every packet, as shown by the "`RECV - / SEND -`" transitions in the Changing and Confirming states.

These state diagrams guarantee safety, but not liveness. Namely, no unexpected or erroneous options will be sent, but option negotiation might not terminate. For example, the following infinite negotiation is legal according to this specification.

```
A > B    Prefer(1)
B > A    Change(2)
A > B    Prefer(1)
B > A    Change(2)...
```

Implementations MAY choose to enforce a maximum length on any negotiation---for example, by resetting the connection when any negotiation lasts more than some maximum time. The DCCP-Reset Reason "Fruitless Negotiation" SHOULD be used to signal that a connection was aborted because of a negotiation that took too long.

In the Changing and Confirming states, the value of the corresponding feature is in flux. DCCP MAY change its behavior in these states---for example, by refusing to send data until reentering a Known state.

6.4. Identification Options

The Identification options provide a way for DCCP endpoints to confirm each others' identities, even after changes of address (Section 10) or long bursts of loss that get the endpoints out of sync (Section 5.2). Again, DCCP as specified here does not provide cryptographic security guarantees, and attackers that can see every packet are still capable of manipulating DCCP connections inappropriately, but the Identification options make it more difficult for some kinds of attacks to succeed.

The Identification option is used to prove an endpoint's identity, while a Challenge option elicits an Identification from the other endpoint. An Identification Regime determines how the Identifications are calculated. In the default MD5 Regime, the calculation involves an MD5 hash over packet data and two Connection Nonces exchanged at the beginning of the connection.

6.4.1. Identification Regime Feature

Identification Regime has feature number 8. The ID Regime feature located at DCCP B specifies the algorithm that DCCP A will use for its Identification options. Each endpoint must keep track of both its ID regime and, via the ID Regime feature, the regime used by the other endpoint.

The value of ID Regime is a two-byte number, so a valid Confirm(ID Regime) option takes exactly four bytes. Change or Prefer options MAY list multiple ID Regimes in descending order of preference. ID Regime defaults to 0, the MD5 Regime. Applications preferring different security guarantees, particularly around mobility issues, may prefer to implement another identification algorithm and assign it a different ID Regime value.

The ID Regime feature is negotiable, so an endpoint can request that the other endpoint use a particular ID Regime, or one of a set of Regimes, by sending a Prefer option. If the endpoints cannot agree on mutually acceptable ID Regimes, the connection SHOULD be reset due to "Fruitless Negotiation".

6.4.2. Connection Nonce Feature

Connection Nonce has feature number 7. The Connection Nonce feature located at DCCP B is the value of DCCP A's connection nonce. Each endpoint SHOULD keep track of both its nonce and, via the Connection Nonce feature, the other endpoint's nonce. Connection Nonces are used by Identification Regime 0.

The Connection Nonce feature takes arbitrary values of at least 4 bytes long. A Change(Connection Nonce) option therefore takes at least 6 bytes. Confirm(Connection Nonce) options MUST NOT contain the relevant value, so a Confirm(Connection Nonce) option takes exactly 2 bytes.

Connection Nonce defaults to a random 8-byte string. To prevent spoofing, this string MUST NOT have any predictable value. For example, it MUST NOT be set deterministically to zero, and it MUST change on every connection.

This feature is non-negotiable.

6.4.3. Identification Option

The Identification option serves as confirmation that a packet was sent by an endpoint involved in the initiation of the DCCP connection. It is permitted in any DCCP packet, but it might not be useful until the endpoints have exchanged security information such as connection nonces. The option takes the following form:

```
+--------+--------+--------+--------+--------+--------
|00101010| Length |  Identification Data ...
+--------+--------+--------+--------+--------+--------
 Type=42
```

The particular data included in an Identification option sent by DCCP A depends on the ID Regime in force for the A-to-B sequence, which is the value of the ID Regime feature located at DCCP B. The remainder of this section describes ID Regime 0, the default MD5 Regime.

The Identification data provided for the MD5 Regime consists of a 16-byte MD5 digest of: the second 32-bit word in the generic DCCP header, including the Sequence Number; the value of the sender's Connection Nonce; and the value of the other endpoint's Connection Nonce, in that order. The total length of the option is therefore 18 bytes. Inclusion of the two Connection Nonces ensures that attackers cannot fake an Identification Option, unless they snooped on the beginning of the connection when nonces are exchanged. (No mechanism protects against snoopers who know Connection Nonces, since DCCP as specified here does not provide strong cryptographic security guarantees; see Section 16.) Inclusion of the sequence number protects against replay attacks within the connection.

To check an Identification option's value, the receiver simply calculates the MD5 digest itself and compares that against the option data. The MD5 calculation can be expensive, so an attacker could conceivably disable a DCCP endpoint by sending it a flood of invalid packets with bad Identification options. Rate limits described in Sections 5.2 and 10 mitigate this issue. The receiver MAY ignore an Identification option if it occurs on a packet that would otherwise be considered valid.

Example C code for constructing the option's value follows:

```
unsigned char *packet_data;
int packet_length;
int id_option_offset; /* offset of option in packet_data */

const unsigned char *my_nonce, *other_nonce;
int my_nonce_length, other_nonce_length;

MD5_CTX md5_context;

MD5_Init(&md5_context);
MD5_Update(&md5_context, packet_data + 4, 4);
MD5_Update(&md5_context, my_nonce, my_nonce_length);
MD5_Update(&md5_context, other_nonce, other_nonce_length);
packet_data[id_option_offset] = 42;   /* option value */
packet_data[id_option_offset+1] = 18; /* option length */
MD5_Final(packet_data + id_option_offset + 2, &md5_context);
```

6.4.4. Challenge Option

This option informs the receiving DCCP that one of its packets was ignored, and that succeeding packets will be ignored until the endpoint sends a correct Identification option. The receiving DCCP SHOULD include an Identification option on the next packet it sends. The option takes the following form:

```
+--------+--------+--------+--------+--------+--------
|00101100| Length |  Identification Data ...
+--------+--------+--------+--------+--------+--------
 Type=44
```

The Identification Data on a packet sent by DCCP B is the same as that for an Identification option sent by DCCP B. The receiver SHOULD ignore a Challenge option, and the packet the Challenge option contains, if the Identification Data is incorrect. The purpose of this mechanism is to prevent denial-of-service attacks where an attacker could cause the receiver to send many packets with expensive-to-compute Identification options, since the receiver MAY ignore Challenge options for some time after receiving an invalid Challenge.

If, after several Challenge options, a DCCP is unable to elicit a valid Identification from its partner, it MAY reset the connection with Reason "Unanswered Challenge".

6.5. Init Cookie Option

This option is permitted in DCCP-Response, DCCP-Data, and DCCP-DataAck messages. The option MAY be returned by the server in a DCCP-Response. If so, then the client MUST echo the same Init Cookie option in its ensuing DCCP-Data or DCCP-DataAck message. The server SHOULD respond to an invalid Init Cookie option by resetting the connection with Reason set to "Bad Init Cookie".

The purpose of this option is to allow a DCCP server to avoid having to hold any state until the three-way connection setup handshake has completed. The server wraps up the service name, server port, and any options it cares about from both the DCCP-Request and DCCP-Response in an opaque cookie. Typically the cookie will be encrypted using a secret known only to the server and include a cryptographic checksum or magic value so that correct decryption can be verified. When the server receives the cookie back in the response, it can decrypt the cookie and instantiate all the state it avoided keeping.

The precise implementation of the Init Cookie does not need to be specified here; since Init Cookies are opaque to the client, there are no interoperability concerns.

```
+--------+--------+--------+--------+--------+--------
|00100100| Length |         Init Cookie Value   ...
+--------+--------+--------+--------+--------+--------
 Type=36
```

6.6. Timestamp Option

This option is permitted in any DCCP packet. The length of the option is 6 bytes.

```
+--------+--------+--------+--------+--------+--------+
|00101000|00000110|          Timestamp Value          |
+--------+--------+--------+--------+--------+--------+
 Type=40  Length=6
```

The four bytes of option data carry the timestamp of this packet in some undetermined form. A DCCP receiving a Timestamp option SHOULD respond with a Timestamp Echo option on the next packet it sends.

6.7. Elapsed Time Option

This option is permitted in any DCCP packet that contains an Acknowledgement Number. It indicates how much time, in milliseconds, has elapsed since the packet being acknowledged---the packet with the given Acknowledgement Number---was received. The option may take up 4 or 6 bytes, depending on how large Elapsed Time is.

```
+--------+--------+--------+--------+
|00101110|00000100|   Elapsed Time  |
+--------+--------+--------+--------+
 Type=46    Len=4

+--------+--------+--------+--------+--------+--------+
|00101110|00000110|            Elapsed Time           |
+--------+--------+--------+--------+--------+--------+
 Type=46    Len=6
```

The option data, Elapsed Time, represents the amount of time, in tenths of milliseconds, elapsed since the packet being acknowledged was received. If Elapsed Time is less than a minute, the first, more parsimonious form of the option SHOULD be used. Elapsed Times of more than 6.5535 seconds MUST be sent using the second form of the option.

Elapsed Time is measured in tenths of milliseconds as a compromise between two conflicting goals: first, to provide enough granularity to reduce aliasing noise when measuring elapsed time over fast LANs; and second, to allow most reasonable elapsed times to fit into two bytes of data.

6.8. Timestamp Echo Option

This option is permitted in any DCCP packet, as long as at least one packet carrying the Timestamp option has been received. The length of the option is between 6 and 10 bytes, depending on whether Elapsed Time is included and how large it is.

```
+--------+--------+--------+--------+--------+--------+
|00101001|00000110|          Timestamp Echo           |
+--------+--------+--------+--------+--------+--------+
 Type=41    Len=6

+--------+--------+------- ... -------+--------+--------+
|00101001|00001000|  Timestamp Echo   |   Elapsed Time  |
+--------+--------+------- ... -------+--------+--------+
 Type=41    Len=8       (4 bytes)

+--------+--------+------- ... -------+------- ... -------+
|00101001|00001010|  Timestamp Echo   |    Elapsed Time   |
+--------+--------+------- ... -------+------- ... -------+
 Type=41   Len=10       (4 bytes)           (4 bytes)
```

The first four bytes of option data, Timestamp Echo, carry a Timestamp Value taken from a preceding received Timestamp option. Usually, this will be the last packet that was received---the packet indicated by the Acknowledgement Number, if any---but it might be a preceding packet.

The Elapsed Time field is similar to the value stored in the Elapsed Time option. If present, it indicates the amount of time elapsed since receiving the packet whose timestamp is being echoed. This time MUST be in tenths of milliseconds. Elapsed Time is meant to help the Timestamp sender separate the network round-trip time from the Timestamp receiver's processing time. This may be particularly important for CCIDs where acknowledgements are sent infrequently, so that there might be considerable delay between receiving a Timestamp option and sending the corresponding Timestamp Echo. A missing Elapsed Time field is equivalent to an Elapsed Time of zero. The smallest version of the option SHOULD be used that can hold the relevant Elapsed Time value.

6.9. Loss Window Feature

Loss Window has feature number 6. The Loss Window feature located at DCCP B is the width of the window DCCP B uses to determine whether packets from DCCP A are valid. Packets outside this window will be dropped by DCCP B as old duplicates or spoofing attempts; see Section 5.2 for more information. DCCP A sends a "Change(Loss Window, W)" option to DCCP B to set DCCP B's Loss Window to W.

The Loss Window feature takes 3-byte integer values, like DCCP sequence numbers. Change and Confirm options for Loss Window are therefore 6 bytes long.

Loss Window defaults to 1000 for new connections. The Loss Window value is the total width of the loss window. The receiver may position the loss window asymmetrically around the greatest sequence number seen---for example, by allocating 1/4 of the loss window width for older sequence numbers and 3/4 of it for newer sequence numbers.

This feature is non-negotiable.

7. Congestion Control IDs

Each congestion control mechanism supported by DCCP is assigned a congestion control identifier, or CCID: a number from 0 to 255. During connection setup, and optionally thereafter, the endpoints negotiate their congestion control mechanisms by negotiating the values for their Congestion Control features. Congestion Control has feature number 1. The feature located at DCCP A is the CCID in use for the A-to-B half-connection. DCCP B sends an "Change(CC, K)" option to DCCP A to ask A to use CCID K for its data packets.

The data byte of Congestion Control feature negotiation options form a list of acceptable CCIDs, sorted in descending order of priority. For example, the option "Change(CC 1, 2, 3)" asks the sender to use CCID 1, although CCIDs 2 and 3 are also acceptable. (This corresponds to the bytes "33, 6, 1, 1, 2, 3": Change option (33), option length (6), feature ID (1), CCIDs (1, 2, 3).) Similarly, "Confirm(CC 1, 2, 3)" tells the receiver that the sender is using CCID 1, but that CCIDs 2 or 3 might also be acceptable.

The CCIDs defined by this document are:

```
CCID   Meaning
----   -------
  0    Reserved
  1    Unspecified Sender-Based Congestion Control
  2    TCP-like Congestion Control
  3    TFRC Congestion Control
```

A new connection starts with CCID 2 for both DCCPs. If this is unacceptable for either DCCP, that DCCP will start in the Unknown state. A DCCP SHOULD NOT send data when its Congestion Control feature is in the Unknown state.

All CCIDs standardized for use with DCCP will correspond to congestion control mechanisms previously standardized by the IETF. We expect that for quite some time, all such mechanisms will be TCP-friendly, but TCP-friendliness is not an explicit DCCP requirement.

A DCCP implementation intended for general use---in a general-purpose operating system kernel, for example---SHOULD implement at least CCIDs 1 and 2. The intent is to make these CCIDs broadly available for interoperability, although any given application might disallow their use via the feature negotiation process.

7.1. Unspecified Sender-Based Congestion Control

CCID 1 denotes an unspecified sender-based congestion control mechanism. Separate features negotiate the corresponding congestion acknowledgement options---for example, Ack Vector. This provides a limited, controlled form of interoperability for new IETF-approved CCIDs.

Implementors MUST NOT use CCID 1 in production environments as a proxy for congestion control mechanisms that have not entered the IETF standards process. We intend that any production use of CCID 1 would have to be explicitly approved first by the IETF. Middleboxes MAY choose to treat the use of CCID 1 as experimental or unacceptable.

For example, say that CCID 98, a new sender-based congestion control mechanism using Ack Vector for acknowledgements, has entered the IETF standards process, and the IETF has approved the use of CCID 1 as a backup for CCID 98. Now, DCCP A, which understands and would like to use CCID 98, is trying to communicate with DCCP B, which doesn't yet know about CCID 98. DCCP A can simply negotiate use of CCID 1 and, separately, negotiate Use Ack Vector. DCCP B will provide the feedback DCCP A requires for CCID 98, namely Ack Vector, without needing to understand the congestion control mechanism in use.

CCID 1 has no sender implementation; it is exclusively meaningful at the receiver to support forward compatibility. The sender always uses a specific congestion control mechanism whose CCID is not 1. However, the code implementing a CCID that requires only generic feedback, such as Ack Vector, MAY add CCID 1 to the list of acceptable CCIDs sent to the receiver (following the actual CCID), facilitating communication with receivers that do not understand the actual CCID.

Any CCID feature negotiation in which the sender proposes the use of CCID 1 without any other CCID is considered erroneous, and SHOULD result in connection reset, with Reason set to "Fruitless Negotiation".

DCCP implementations MAY provide APIs that allow applications to suggest preferred CCIDs for sending and receiving data. Any such API MUST NOT allow sending applications to suggest CCID 1; again, CCID 1 will be suggested when appropriate by the code implementing the preferred CCID. In contrast, APIs SHOULD let applications allow or prevent the use of CCID 1 for receiving.

7.2. TCP-like Congestion Control

CCID 2 denotes Additive Increase, Multiplicative Decrease (AIMD) congestion control with behavior modelled directly on TCP, including congestion window, slow start, timeouts, and so forth. CCID 2 is further described in [CCID 2 PROFILE].

7.3. TFRC Congestion Control

CCID 3 denotes TCP-Friendly Rate Control, an equation-based rate-controlled congestion control mechanism. CCID 3 is further described in [CCID 3 PROFILE].

7.4. CCID-Specific Options and Features

Option and feature numbers 128 through 255 are available for CCID-specific use. CCIDs may often need new option types---for communicating acknowledgement or rate information, for example. CCID-specific option types let them create options at will without polluting the global option space. Option 128 might have different meanings on a half-connection using CCID 4 and a half-connection using CCID 8. CCID-specific options and features will never conflict with global options introduced by later versions of this specification.

Any packet may contain information meant for either half-connection, so CCID-specific option and feature numbers explicitly signal the half-connection to which they apply. Option numbers 128 through 191 are for options sent from the HC-Sender to the HC-Receiver; option numbers 192 through 255 are for options sent from the HC-Receiver to the HC-Sender. Similarly, feature numbers 128 through 191 are for features located at the HC-Sender; feature numbers 192 through 255 are for features located at the HC-Receiver. (Change options for a feature are sent to the feature location; Prefer and Confirm options are sent from the feature location. Thus, Change(128) options are sent by the HC-Receiver by definition, while Change(192) options are sent by the HC-Sender.)

For example, consider a DCCP connection where the A-to-B half-connection uses CCID 4 and the B-to-A half-connection uses CCID 5. Here is how a sampling of CCID-specific options and features are assigned to half-connections:

Packet	Option	Relevant Half-conn.	Relevant CCID
A > B	128	A-to-B	4
A > B	192	B-to-A	5
A > B	Change(128, ...)	B-to-A	5
A > B	Prefer(128, ...)	A-to-B	4
A > B	Confirm(128, ...)	A-to-B	4
A > B	Change(192, ...)	A-to-B	4
A > B	Prefer(192, ...)	B-to-A	5
A > B	Confirm(192, ...)	B-to-A	5

CCID-specific options and features have no clear meaning when the relevant CCID is in flux. A DCCP SHOULD respond to CCID-specific options and features with Ignored options during those times.

8. Acknowledgements

Congestion control requires receivers to transmit information about packet losses and ECN marks to senders. DCCP receivers MUST report all congestion they see, as defined by the relevant CCID profile. Each CCID says when acknowledgements should be sent, what options they must use, how they should be congestion controlled, and so on.

Most acknowledgements use DCCP options. For example, on a half-connection with CCID 2 (TCP-like), the receiver reports acknowledgement information using the Ack Vector option. This section describes common acknowledgement options and shows how acks using those options will commonly work. Full descriptions of the acknowledgement mechanisms used for each CCID are laid out in the CCID profile specifications.

Acknowledgement options, such as Ack Vector, generally depend on the DCCP Acknowledgement Number, and are thus only allowed on packet types that carry that number (all packets except DCCP-Request and DCCP-Data). However, detailed acknowledgement options are not generally necessary on DCCP-Resets.

8.1. Acks of Acks and Unidirectional Connections

DCCP was designed to work well for both bidirectional and unidirectional flows of data, and for connections that transition between these states. However, acknowledgements required for a unidirectional connection are very different from those required for a bidirectional connection. In particular, unidirectional connections need to worry about acks of acks.

The ack-of-acks problem arises because some acknowledgement mechanisms are reliable. For example, an HC-Receiver using CCID 2, TCP-like Congestion Control, sends Ack Vectors containing completely reliable acknowledgement information. The HC-Sender should occasionally inform the HC-Receiver that it has received an ack. If it did not, the HC-Receiver might resend complete Ack Vector information, going back to the start of the connection, with every DCCP-Ack packet! However, note that acks-of-acks need not be reliable themselves: when an ack-of-acks is lost, the HC-Receiver will simply maintain old acknowledgement-related state for a little longer. Therefore, there is no need for acks-of-acks-of-acks.

When communication is bidirectional, any required acks-of-acks are automatically contained in normal acknowledgements for data packets. On a unidirectional connection, however, the receiver DCCP sends no data, so the sender would not normally send acknowledgements. Therefore, the CCID in force on that half-connection must explicitly say whether, when, and how the HC-Sender should generate acks-of-acks.

For example, consider a bidirectional connection where both half-connections use the same CCID (either 2 or 3), and where DCCP B goes "quiescent". This means that the connection becomes unidirectional: DCCP B stops sending data, and sends only sends DCCP-Ack packets to DCCP A. For CCID 2, TCP-like Congestion Control, DCCP B uses Ack Vector to reliably communicate which packets it has received. As described above, DCCP A must occasionally acknowledge a pure acknowledgement from DCCP B, so that DCCP B can free old Ack Vector state. For instance, DCCP A might send a DCCP-DataAck packet every now and then, instead of DCCP-Data. In contrast, for CCID 3, TFRC Congestion Control, DCCP B's acknowledgements generally need not be reliable, since they contain cumulative loss rates; TFRC works even if every DCCP-Ack is lost. Therefore, DCCP A need never acknowledge an acknowledgement.

When communication is unidirectional, a single CCID---in the example, the A-to-B CCID---controls both DCCPs' acknowledgements, in terms of their content, their frequency, and so forth. For bidirectional connections, the A-to-B CCID governs DCCP B's acknowledgements (including its acks of DCCP A's acks), while the B-to-A CCID governs DCCP A's acknowledgements.

DCCP A switches its ack pattern from bidirectional to unidirectional when it notices that DCCP B has gone quiescent. It switches from unidirectional to bidirectional when it must acknowledge even a single DCCP-Data or DCCP-DataAck packet from DCCP B. (This includes the case where a single DCCP-Data or DCCP-DataAck packet was lost in transit, which is detectable using the # NDP field in the DCCP packet header.)

Each CCID defines how to detect quiescence on that CCID, and how that CCID handles acks-of-acks on unidirectional connections. The B-to-A CCID defines when DCCP B has gone quiescent. Usually, this happens when a period has passed without B sending any data packets. For CCID 2, this period is roughly two round-trip times. The A-to-B CCID defines how DCCP A handles acks-of-acks once DCCP B has gone quiescent.

8.2. Ack Piggybacking

Acknowledgements of A-to-B data MAY be piggybacked on data sent by DCCP B, as long as that does not delay the acknowledgement longer than the A-to-B CCID would find acceptable. However, data acknowledgements often require more than 4 bytes to express. A large set of acknowledgements prepended to a large data packet might exceed the path's MTU. In this case, DCCP B SHOULD send separate DCCP-Data and DCCP-Ack packets, or wait, but not too long, for a smaller datagram.

Piggybacking is particularly common at DCCP A when the B-to-A half-connection is quiescent---that is, when DCCP A is just acknowledging DCCP B's acknowledgements, as described above. There are three reasons to acknowledge DCCP B's acknowledgements: to allow DCCP B to free up information about previously acknowledged data packets from A; to shrink the size of future acknowledgements; and to manipulate the rate future acknowledgements are sent. Since these are secondary concerns, DCCP A can generally afford to wait indefinitely for a data packet to piggyback its acknowledgement onto.

Any restrictions on ack piggybacking are described in the relevant CCID's profile.

8.3. Ack Ratio Feature

Ack Ratio provides a common mechanism by which CCIDs that clock acknowledgements off of data packets can perform rudimentary congestion control on the acknowledgement stream. CCID 2, TCP-like Congestion Control, uses Ack Ratio to limit the rate of its acknowledgement stream, for example. Some CCIDs ignore Ack Ratio, performing congestion control on acknowledgements in some other way.

Ack Ratio has feature number 3. The Ack Ratio feature located at DCCP B equals the ratio of data packets sent by DCCP A to acknowledgement packets sent back by DCCP B. For example, if it is set to four, then DCCP B will send at least one acknowledgement packet for every four data packets DCCP A sends. DCCP A sends a "Change(Ack Ratio)" option to DCCP B to change DCCP B's ack ratio.

An Ack Ratio option contains two bytes of data: a sixteen-bit integer representing the ratio. A new connection starts with Ack Ratio 2 for both DCCPs.

This feature is non-negotiable.

8.4. Use Ack Vector Feature

The Use Ack Vector feature lets DCCPs negotiate whether they should use Ack Vector options to report congestion. Ack Vector provides detailed loss information, and lets senders report back to their applications whether particular packets were dropped. Use Ack Vector is mandatory for some CCIDs, and optional for others.

Use Ack Vector has feature number 4. The Use Ack Vector feature located at DCCP B specifies whether DCCP B MUST use Ack Vector options on its acknowledgements to DCCP A, although DCCP B MAY send Ack Vector options even when Use Ack Vector is false. DCCP A sends a "Change(Use Ack Vector, 1)" option to DCCP B to ask B to send Ack Vector options as part of its acknowledgement traffic.

Use Ack Vector feature values are a single byte long. The receiver MUST send Ack Vector options if this byte is nonzero. A new connection starts with Use Ack Vector 0 for both DCCPs.

8.5. Ack Vector Options

The Ack Vector gives a run-length encoded history of data packets received at the client. Each byte of the vector gives the state of that data packet in the loss history, and the number of preceding packets with the same state. The option's data looks like this:

```
+--------+--------+--------+--------+--------+--------
|001001??| Length |SSLLLLLL|SSLLLLLL|SSLLLLLL|  ...
+--------+--------+--------+--------+--------+--------
Type=37/38         \___________ Vector ___________...
```

The two Ack Vector options (option types 37 and 38) differ only in the values they imply for ECN Nonce Echo. Section 9.2 describes this further.

The vector itself consists of a series of bytes, each of whose encoding is:

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|St | Run Length|
+-+-+-+-+-+-+-+-+
```

St[ate]: 2 bits

Run Length: 6 bits

State occupies the most significant two bits of each byte, and can have one of four values:

0 Packet received (and not ECN marked).

1 Packet received ECN marked.

2 Reserved.

3 Packet not yet received.

The first byte in the first Ack Vector option refers to the packet indicated in the Acknowledgement Number; subsequent bytes refer to older packets. (Ack Vector MUST NOT be sent on DCCP-Data and DCCP-Request packets, which lack an Acknowledgement Number.) If an Ack Vector contains the decimal values 0,192,3,64,5 and the Acknowledgement Number is decimal 100, then:

Packet 100 was received (Acknowledgement Number 100, State 0, Run Length 0).

Packet 99 was lost (State 3, Run Length 0).

Packets 98, 97, 96 and 95 were received (State 0, Run Length 3).

Packet 94 was ECN marked (State 1, Run Length 0).

Packets 93, 92, 91, 90, 89, and 88 were received (State 0, Run Length 5).

Run lengths of more than 64 must be encoded in multiple bytes. A single Ack Vector option can acknowledge up to 16192 data packets. Should more packets need to be acknowledged than can fit in 253 bytes of Ack Vector, then multiple Ack Vector options can be sent. The second Ack Vector option will begin where the first Ack Vector option left off, and so forth.

Ack Vector states are subject to two general constraints. (These principles SHOULD also be followed for other acknowledgement mechanisms; referring to Ack Vector states simplifies their explanation.)

(1) Packets reported as State 0 or State 1 MUST have been processed by the receiving DCCP stack. In particular, their options must have been processed. Any data on the packet need not have been delivered to the receiving application; in fact, the data may have been dropped.

(2) Packets reported as State 3 MUST NOT have been received by DCCP. Feature negotiations and options on such packets MUST NOT have been processed, and the Acknowledgement Number MUST NOT correspond to such a packet.

Packets dropped in the application's receive buffer SHOULD be reported as Received or Received ECN Marked (States 0 and 1), depending on their ECN state; such packets' ECN Nonces MUST be included in the Nonce Echo. The Data Dropped option informs the sender that some packets reported as received actually had their payloads dropped.

One or more Ack Vector options that, together, report the status of more packets than have actually been sent SHOULD be considered invalid. The receiving DCCP SHOULD either ignore the options or reset the connection with Reason set to "Option Error". Packets whose status has not reported by any Ack Vector option SHOULD be treated as "not yet received" (State 3) by the sender.

8.5.1. Ack Vector Consistency

A DCCP sender will commonly receive multiple acknowledgements for some of its data packets. For instance, an HC-Sender might receive two DCCP-Acks with Ack Vectors, both of which contained information about sequence number 24. (Because of cumulative acking, information about a sequence number is repeated in every ack until the HC-Sender acknowledges an ack. Perhaps the HC-Receiver is sending acks faster than the HC-Sender is acknowledging them.) In a perfect world, the two Ack Vectors would always be consistent. However, there are many reasons why they might not be:

- The HC-Receiver received packet 24 between sending its acks, so the first ack said 24 was not received (State 3) and the second said it was received or ECN marked (State 0 or 1).
- The HC-Receiver received packet 24 between sending its acks, and the network reordered the acks. In this case, the packet will appear to transition from State 0 or 1 to State 3.
- The network duplicated packet 24, and one of the duplicates was ECN marked. This might show up as a transition between States 0 and 1.

To cope with these situations, HC-Sender DCCP implementations SHOULD combine multiple received Ack Vector states according to this table:

```
                 Received State
                   0   1   3
                 +---+---+---+
               0 | 0 | 1 | 0 |
        Old      +---+---+---+
               1 | 1 | 1 | 1 |
       State     +---+---+---+
               3 | 0 | 1 | 3 |
                 +---+---+---+
```

To read the table, choose the row corresponding to the packet’s old state and the column corresponding to the packet’s state in the newly received Ack Vector, then read the packet’s new state off the table. The table is symmetric about the main diagonal, so it is indifferent to ack reordering.

This table defines how the HC-Sender should react to received Ack Vector states. This is equivalent to how the HC-Receiver should collect information about received packets, with two symmetric exceptions: when one State is 0 (received non-marked) and the other is 1 (received ECN marked). According to the table, the HC-Sender should react to this combination of Ack Vector information as if only State 1 had been reported. But what state should the HC-Receiver report in Ack Vector if two duplicates are received for a packet, and only one is ECN marked? We explicitly allow the HC-Receiver to report the combination as State 0 (received non-marked) or State 1. After all, one duplicate was non-marked, and depending on how much state the HC-Receiver keeps about packets it receives, it might be impossible to change a packet from State 0 to State 1 and preserve correct ECN Nonce Echo information.

A HC-Sender MAY choose to throw away old information gleaned from the HC-Receiver’s Ack Vectors, in which case it MUST ignore newly received acknowledgements from the HC-Receiver for those old packets. It is often kinder to save recent Ack Vector information for a while, so that the HC-Sender can undo its reaction to presumed congestion when a "lost" packet unexpectedly shows up (the transition from State 3 to State 0).

8.5.2. Ack Vector Coverage

We can divide the packets that have been sent from an HC-Sender to an HC-Receiver into four roughly contiguous groups. From oldest to youngest, these are:

(1) Packets already acknowledged by the HC-Receiver, where the HC-Receiver knows that the HC-Sender has definitely received the acknowledgements.

(2) Packets already acknowledged by the HC-Receiver, where the HC-Receiver cannot be sure that the HC-Sender has received the acknowledgements.

(3) Packets not yet acknowledged by the HC-Receiver.

(4) Packets not yet received by the HC-Receiver.

The union of groups 2 and 3 is called the Unacknowledged Window. Generally, every Ack Vector generated by the HC-Receiver will cover the whole Unacknowledged Window: Ack Vector acknowledgements are cumulative. (This simplifies Ack Vector maintenance at the HC-Receiver; see Section 8.9, below.) As packets are received, this window both grows on the right and shrinks on the left. It grows because there are more packets, and shrinks because the data packets' Acknowledgement Numbers will acknowledge previous acknowledgements, moving packets from group 2 into group 1.

8.6. Slow Receiver Option

An HC-Receiver sends the Slow Receiver option to its sender to indicate that it is having trouble keeping up with the sender's data. The HC-Sender SHOULD NOT increase its sending rate for approximately one round-trip time after seeing a packet with a Slow Receiver option. However, the Slow Receiver option does not indicate congestion, and the HC-Sender need not reduce its sending rate. (If necessary, the receiver can force the sender to slow down by dropping packets or reporting false ECN marks.) APIs SHOULD let receiver applications set Slow Receiver, and sending applications determine whether or not their receivers are Slow.

The Slow Receiver option takes just one byte:

```
+--------+
|00000010|
+--------+
 Type=2
```

Slow Receiver does not specify why the receiver is having trouble keeping up with the sender. Possible reasons include lack of buffer space, CPU overload, and application quotas. A sending application might react to Slow Receiver by reducing its sending rate or by switching to a lossier compression algorithm. However, a smart sender might actually *increase* its sending rate in response to Slow Receiver, by switching to a less-compressed sending format. (A highly-compressed data format might overwhelm a slow CPU more seriously than the higher memory requirements of a less-compressed data format.) This tension between transfer size (less compression means more congestion) and processing speed (less compression means less processing) cannot be resolved in general.

Slow Receiver implements a portion of TCP's receive window functionality. We believe receiver operating systems and applications will find it much easier to send Slow Receiver when appropriate than they currently find it to correctly set a TCP receive window.

8.7. Data Dropped Option

The Data Dropped option indicates that some packets reported as received actually had their data dropped before it reached the application. The sender's congestion control mechanism MAY react to data-dropped packets; such responses MAY be less severe than responses triggered by a lost or marked packet. (For instance, a windowed mechanism might subtract a constant value from its congestion window, rather than cut it in half.) When ECN-marked packets are included in Data Dropped, the sender's congestion control mechanism MUST react to the ECN marks as usual.

The option's data looks like this:

```
+--------+--------+--------+--------+--------+--------
|00100111| Length | Block  | Block  | Block  |  ...
+--------+--------+--------+--------+--------+--------
 Type=39          \___________ Vector ___________ ...
```

The vector itself consists of a series of bytes, called Blocks, each of whose encoding corresponds to one of these choices:

```
 0 1 2 3 4 5 6 7                  0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+                +-+-+-+-+-+-+-+-+
|0| Run Length  |       or       |1|Dr St|Run Len|
+-+-+-+-+-+-+-+-+                +-+-+-+-+-+-+-+-+
  Normal Block                       Drop Block
```

The first byte in the first Data Dropped option refers to the packet indicated in the Acknowledgement Number; subsequent bytes refer to older packets. (Data Dropped MUST NOT be sent on DCCP-Data or DCCP-Request packets, which lack an Acknowledgement Number.) Normal Blocks, which have high bit 0, indicate that any received packets in the Run Length had their data delivered to the application. Drop Blocks, which have high bit 1, indicate that received packets in the Run Len[gth] were not delivered as usual. The 3-bit Dr[op] St[ate] field says what happened; generally, no data from that packet reached the application. Packets reported as "not yet received" MUST be included in Normal Blocks; packets not covered by any Data Dropped option are treated as if they were in a Normal Block. Defined Drop States for Drop Blocks are:

0 Packet data dropped due to protocol constraints. For example, the data was included on a DCCP-Request packet, and the receiving application does not allow that piggybacking; or the data was sent during an important feature negotiation.

1 Packet data dropped in the receive buffer.

2 Packet data dropped due to corruption.

3 Packet data corrupted, but delivered to the application anyway.

4 Packet data dropped because the application is no longer listening.

5-7 Reserved.

For example, if a Data Dropped option contains the decimal values 0,144,3,146, the Acknowledgement Number is 100, and an Ack Vector reported all packets as received, then:

Packet 100 was received (Acknowledgement Number 100, Normal Block, Run Length 0).

Packet 99 was dropped in the receive buffer (Drop Block, Drop State 1, Run Length 0).

Packets 98, 97, 96, and 95 were received (Normal Block, Run Length 3).

Packets 95, 94, and 93 were dropped in the receive buffer (Drop Block, Drop State 1, Run Length 2).

Run lengths of more than 128 (for Normal Blocks) or 16 (for Drop Blocks) must be encoded in multiple Blocks. A single Data Dropped option can acknowledge up to 32384 Normal Block data packets, although the receiver SHOULD NOT send a Data Dropped option when all relevant packets fit into Normal Blocks. Should more packets need to be acknowledged than can fit in 253 bytes of Data Dropped, then multiple Data Dropped options can be sent. The second option will begin where the first option left off, and so forth.

One or more Data Dropped options that, together, report the status of more packets than have been sent, or that change the status of a packet, or that disagree with Ack Vector or equivalent options (by reporting a "not yet received" packet as "dropped in the receive buffer", for example), SHOULD be considered invalid. The receiving DCCP SHOULD respond to invalid Data Dropped options by ignoring them or by resetting the connection with Reason set to "Option Error".

Drop State 4 ("application no longer listening") means the application running at the endpoint that sent the option is no longer listening for data. For example, a server might close its receiving half-connection to new data after receiving a complete request from the client. This would limit the amount of state the server would expend on incoming data, and thus reduce the potential damage from certain denial-of-service attacks. A Data Dropped option containing Drop State 4 SHOULD be sent whenever received data is ignored due to a non-listening application. Once a DCCP reports Drop State 4 for a packet, it SHOULD report Drop State 4 for every succeeding data packet on that half-connection; once a DCCP receives a Drop State 4 report, it SHOULD expect that no more data will ever be delivered to the other endpoint's application. A DCCP receiving Drop State 4 MAY report this event to the application. (Previous versions of this specification used a "Buffer Closed" option instead of Drop State 4.)

8.8. Payload Checksum Option

The Payload Checksum option holds the 16 bit one's complement of the one's complement sum of all 16 bit words in the DCCP payload (the data contained in a DCCP-Request, DCCP-Response, DCCP-Data, DCCP-DataAck, or DCCP-Move packet). When combined with a Checksum Length of less than 15, this lets DCCP distinguish between corruption in a packet's payload and corruption in its header. Corrupted-header packets MUST be treated as dropped by the network, while corrupted-payload packets MAY be treated differently; for example, the sender's response to corruption might be less stringent than its response to congestion. A low Checksum Length lets DCCP process packets with valid headers, even if the payload is corrupt, avoiding the congestion response to corruption. The Payload Checksum option then lets DCCP detect payload corruption, and therefore avoid delivering bad data to the application.

The option's data looks like this:

```
+--------+--------+--------+--------+
|00101101|00000100|     Checksum    |
+--------+--------+--------+--------+
 Type=45  Length=4
```

The receiving DCCP MUST check the Payload Checksum's value against the actual payload checksum. If the values differ, the packet's data SHOULD be dropped, and reported as dropped due to corruption (Drop State 2) using a Data Dropped option (Section 8.7). Optionally, DCCP MAY provide an API through which the receiving application could request delivery of known-corrupt data. When that API is active, the packet's data SHOULD be delivered, but reported as delivered corrupt (Drop State 3) using a Data Dropped option. In either case, the packet will be reported as Received or Received ECN Marked by Ack Vector or equivalent options.

See Section 18.1 for a discussion of the issues related to the use of this option.

8.9. Ack Vector Implementation Notes

This section discusses particulars of DCCP acknowledgement handling, in the context of an abstract implementation for Ack Vector. It is informative rather than normative.

The first part of our implementation runs at the HC-Receiver, and therefore acknowledges data packets. It generates Ack Vector options. The implementation has the following characteristics:

- At most one byte of state per acknowledged packet.
- O(1) time to update that state when a new packet arrives (normal case).
- Cumulative acknowledgements.
- Quick removal of old state.

The basic data structure is a circular buffer containing information about acknowledged packets. Each byte in this buffer contains a state and run length; the state can be 0 (packet received), 1 (packet ECN marked), or 3 (packet not yet received). The live portion of the buffer is marked off by head and tail pointers, each marked with the HC-Sender sequence number to which it corresponds. The buffer also stores a single-bit ECN Nonce Echo, which equals the one-bit sum of the ECN Nonces received on state-0 packets. The buffer grows from right to left. For example:

```
+---------------------------------------------------------------------+
|S,L|S,L|S,L|S,L|   |   |   |   |S,L|S,L|S,L|S,L|S,L|S,L|S,L|S,L|
+---------------------------------------------------------------------+
              ^                        ^
      Tail, seqno = T         Head, seqno = H     ECN Nonce Echo = E

             <=== Head and Tail move this way <===
```

Each '`S,L`' represents a State/Run length byte. We will draw these buffers showing only their live portion; for example, here is another representation for the buffer above:

```
   +-----------------------------------------------+
 H |S,L|S,L|S,L|S,L|S,L|S,L|S,L|S,L|S,L|S,L|S,L|S,L| T   ENE[E]
   +-----------------------------------------------+
```

This smaller Example Buffer contains actual data.

```
        +---------------------------+
     10 |0,0|3,0|3,0|3,0|0,4|1,0|0,0| 0   ENE[1]   [Example Buffer]
        +---------------------------+
```

In concrete terms, its meaning is as follows:

Packet 10 was received. (The head of the buffer has sequence number 10, state 0, and run length 0.)

Packets 9, 8, and 7 have not yet been received. (The three bytes preceding the head each have state 3 and run length 0.)

Packets 6, 5, 4, 3, and 2 were received.

Packet 1 was ECN marked.

Packet 0 was received.

The one-bit sum of the ECN Nonces on packets 10, 6, 5, 4, 3, 2, and 0 equals 1.

8.9.1. New Packets

When a packet arrives whose sequence number is larger than any in the buffer, the HC-Receiver simply moves the Head pointer to the left, increases the head sequence number, and stores a byte representing the packet into the buffer. For example, if HC-Sender packet 11 arrived ECN marked, the Example Buffer above would enter this new state (the change is marked with stars):

```
        +***-------------------------------+
     11 |1,0|0,0|3,0|3,0|3,0|0,4|1,0|0,0| 0   ENE[1]
        +***-------------------------------+
```

If the packet's state equals the state at the head of the buffer, the HC-Receiver may choose to increment its run length (up to the maximum). For example, if HC-Sender packet 11 arrived without ECN marking and with ECN Nonce 0, the Example Buffer might enter this state instead:

```
            +--*------------------------+
         11 |0,1|3,0|3,0|3,0|0,4|1,0|0,0| 0   ENE[1]
            +--*------------------------+
```

Of course, the new packet's sequence number might not equal the expected sequence number. In this case, the HC-Receiver should enter the intervening packets as State 3. If several packets are missing, the HC-Receiver may prefer to enter multiple bytes with run length 0, rather than a single byte with a larger run length; this simplifies table updates when one of the missing packets arrives. For example, if HC-Sender packet 12 arrived with ECN Nonce 1, the Example Buffer would enter this state:

```
    +*******-------------------------------+           *
 12 |0,0|3,0|0,1|3,0|3,0|3,0|0,4|1,0|0,0| 0   ENE[0]
    +*******-------------------------------+           *
```

When a new packet's sequence number is less than the head sequence number, the HC-Receiver should scan the table for the byte corresponding to that sequence number. (Slightly more complex indexing structures could reduce the complexity of this scan.) Assume that the sequence number was previously lost (State 3), and that it was stored in a byte with run length 0. Then the HC-Receiver can simply change the byte's state. For example, if HC-Sender packet 8 was received with ECN Nonce 0, the Example Buffer would enter this state:

```
       +--------*------------------+
    10 |0,0|3,0|0,0|3,0|0,4|1,0|0,0| 0   ENE[1]
       +--------*------------------+
```

If the packet is not marked as lost, or if its sequence number is not contained in the table, the packet is probably a duplicate, and should be ignored. (The new packet's ECN marking state might differ from the state in the buffer; Section 8.5.1 describes what to do then.) If the packet's corresponding buffer byte has a non-zero run length, then the buffer might need be reshuffled to make space for one or two new bytes.

Of course, the circular buffer may overflow, either when the HC-Sender is sending data at a very high rate, when the HC-Receiver's acknowledgements are not reaching the HC-Sender, or when the HC-Sender is forgetting to acknowledge those acks (so the HC-Receiver is unable to clean up old state). In this case, the HC-Receiver should either compress the buffer, transfer its state to a larger buffer, or drop all received packets, without processing them whatsoever, until its buffer shrinks again.

8.9.2. Sending Acknowledgements

Whenever the HC-Receiver needs to generate an acknowledgement, the buffer's contents can simply be copied into one or more Ack Vector options. Copied Ack Vectors might not be maximally compressed; for example, the Example Buffer above contains three adjacent 3,0 bytes that could be combined into a single 3,2 byte. The HC-Receiver might, therefore, choose to compress the buffer in place before sending the option, or to compress the buffer while copying it; either operation is simple.

Every acknowledgement sent by the HC-Receiver SHOULD include the entire state of the buffer. That is, acknowledgements are cumulative.

The HC-Receiver should store information about each acknowledgement it sends in another buffer. Specifically, for every acknowledgement it sends, the HC-Receiver should store:

- The HC-Receiver sequence number it used for the ack packet.
- The HC-Sender sequence number it acknowledged (that is, the packet's Acknowledgement Number). Since acknowledgements are cumulative, this single number completely specifies the set of HC-Sender packets acknowledged by this ack packet.

8.9.3. Clearing State

Some of the HC-Sender's packets will include acknowledgement numbers, which ack the HC-Receiver's acknowledgements. When such an ack is received, the HC-Receiver simply finds the HC-Sender sequence number corresponding to that acked HC-Receiver packet, and moves the buffer's Tail pointer up to that sequence number. (It may choose to keep some older information, in case a lost packet shows up late.) For example, say that the HC-Receiver storing the Example Buffer had sent two acknowledgements already:

(1) HC-Receiver Ack 59 acknowledged HC-Sender Seq 3 with ECN Nonce Echo 1.

(2) HC-Receiver Ack 60 acknowledged HC-Sender Seq 10 with ECN Nonce Echo 0.

Say the HC-Receiver then received a DCCP-DataAck packet from the HC-Sender with Acknowledgement Number 59. This informs the HC-Receiver that the HC-Sender received, and processed, all the information in HC-Receiver packet 59. This packet acknowledged HC-Sender packet 3, so the HC-Sender has now received HC-Receiver's acknowledgements for packets 0, 1, 2, and 3. The Example Buffer should enter this state:

```
       +------------------*+ *        *
    10 |0,0|3,0|3,0|3,0|0,2| 4   ENE[0]
       +------------------*+ *        *
```

The tail byte's run length was adjusted, since packet 3 was in the middle of that byte. The new ECN Nonce Echo field equals the exclusive-or of the old field, and the ECN Nonce Echo reported with the relevant acknowledgement. The HC-Receiver can also throw away stored information about HC-Receiver Ack 59.

A careful implementation might try to ensure reasonable robustness to reordering. Suppose that the Example Buffer is as before, but that packet 9 now arrives, out of sequence. The buffer would enter this state:

```
       +----*----------------------+
    10 |0,0|0,0|3,0|3,0|0,4|1,0|0,0| 0     ENE[1]
       +----*----------------------+
```

The danger is that the HC-Sender might acknowledge the P2's previous acknowledgement (with sequence number 60), which says that Packet 9 was not received, before the HC-Receiver has a chance to send a new acknowledgement saying that Packet 9 actually was received. Therefore, when packet 9 arrived, the HC-Receiver might modify its acknowledgement record to:

(1) HC-Receiver Ack 59 acknowledged HC-Sender Seq 3 with ECN Nonce Echo 1.

(2) HC-Receiver Ack 60 also acknowledged HC-Sender Seq 3 with ECN Nonce Echo 1.

That is, Ack 60 is now treated like a duplicate of Ack 59. This would prevent the Tail pointer from moving past packet 9 until the HC-Receiver knows that the HC-Sender has seen an Ack Vector indicating that packet's arrival.

8.9.4. Processing Acknowledgements

When the HC-Sender receives an acknowledgement, it generally cares about the number of packets that were dropped and/or ECN marked. It simply reads this off the Ack Vector. Additionally, it may check the ECN Nonce for correctness. (As described in Section 8.5.1, it may want to keep more detailed information about acknowledged packets in case packets change states between acknowledgements, or in case the application queries whether a packet arrived.)

The HC-Sender must also acknowledge the HC-Receiver's acknowledgements so that the HC-Receiver can free old Ack Vector state. (Since Ack Vector acknowledgements are reliable, the HC-Receiver must maintain and resend Ack Vector information until it is sure that the HC-Sender has received that information.) A simple algorithm suffices: since Ack Vector acknowledgements are cumulative, a single acknowledgement number tells HC-Receiver how much ack information has arrived. Assuming that the HC-Receiver sends no data, the HC-Sender can simply ensure that at least once a round-trip time, it sends a DCCP-DataAck packet acknowledging the latest DCCP-Ack packet it has received. Of course, the HC-Sender only needs to acknowledge the HC-Receiver's acknowledgements if the HC-Sender is also sending data. If the HC-Sender is not sending data, then the HC-

Receiver's Ack Vector state is stable, and there is no need to shrink it. The HC-Sender must watch for drops and ECN marks on received DCCP-Ack packets so that it can adjust the HC-Receiver's ack-sending rate---for example, with Ack Ratio---in response to congestion.

If the other half-connection is not quiescent---that is, the HC-Receiver is sending data to the HC-Sender, possibly using another CCID---then the acknowledgements on that half-connection are sufficient for the HC-Receiver to free its state.

9. Explicit Congestion Notification

The DCCP protocol is fully ECN-aware. Each CCID specifies how its endpoints respond to ECN marks. Furthermore, DCCP, unlike TCP, allows senders to control the rate at which acknowledgements are generated (with options like Ack Ratio); this means that acknowledgements are generally congestion-controlled, and may have ECN-Capable Transport set.

A CCID profile describes how that CCID interacts with ECN, both for data traffic and pure-acknowledgement traffic. A sender SHOULD set ECN-Capable Transport on its packets whenever the receiver has its ECN Capable feature turned on and the relevant CCID allows it, unless the sending application indicates that ECN should not be used.

The rest of this section describes the ECN Capable feature and the interaction of the ECN Nonce with acknowledgement options such as Ack Vector.

9.1. ECN Capable Feature

The ECN Capable feature lets a DCCP inform its partner that it cannot read ECN bits from received IP headers, so the partner must not set ECN-Capable Transport on its packets.

ECN Capable has feature number 2. The ECN Capable feature located at DCCP A indicates whether or not A can successfully read ECN bits from received frames' IP headers. (This is independent of whether it can set ECN bits on sent frames.) DCCP A sends a "Prefer(ECN Capable, 0)" option to DCCP B to inform B that A cannot read ECN bits.

An ECN Capable feature contains a single byte of data. ECN capability is on if and only if this byte is nonzero.

A new connection starts with ECN Capable 1 (that is, ECN capable) for both DCCPs. If a DCCP is not ECN capable, it MUST send "Prefer(ECN Capable, 0)" options to the other endpoint until acknowledged (by "Change(ECN Capable, 0)") or the connection closes. Furthermore, it MUST NOT accept any data until the other endpoint sends "Change(ECN Capable, 0)". It SHOULD send Data Dropped options on its acknowledgements, with Drop State 0 ("protocol constraints"), if the other endpoint does send data inappropriately.

9.2. ECN Nonces

Congestion avoidance will not occur, and the receiver will sometimes get its data faster, when the sender is not told about any congestion events. Thus, the receiver has some incentive to falsify acknowledgement information, reporting that marked or dropped packets were actually received unmarked. This problem is more serious with DCCP than with TCP, since TCP provides reliable transport: it is more difficult with TCP to lie about lost packets without breaking the application.

ECN Nonces are a general mechanism to prevent ECN cheating (or loss cheating). Two values for the two-bit ECN header field indicate ECN-Capable Transport, 01 and 10. The second code point, 10, is the ECN Nonce. In general, a protocol sender chooses between these code points randomly on its output packets, remembering the sequence it chose. The protocol receiver reports, on every acknowledgement, the number of ECN Nonces it has received thus far. This is called the ECN Nonce Echo. Since ECN marking and packet dropping both destroy the ECN Nonce, a receiver that lies about an ECN mark or packet drop has a 50% chance of guessing right and avoiding discipline. The sender may react punitively to an ECN Nonce mismatch, possibly up to dropping the connection. The ECN Nonce Echo field need not be an integer; one bit is enough to catch 50% of infractions.

In DCCP, the ECN Nonce Echo field is encoded in acknowledgement options. For example, the Ack Vector option comes in two forms, Ack Vector [Nonce 0] (option 37) and Ack Vector [Nonce 1] (option 38), corresponding to the two values for a one-bit ECN Nonce Echo. The Nonce Echo for a given Ack Vector equals the one-bit sum (exclusive-or, or parity) of ECN nonces for packets reported by that Ack Vector as received and not ECN marked. Thus, only packets marked as State 0 matter for this calculation (that is, valid received packets that were not ECN marked). Every Ack Vector option is detailed enough for the sender to determine what the Nonce Echo should have been. It can check this calculation against the actual Nonce Echo, and complain if there is a mismatch.

(The Ack Vector could conceivably report every packet's ECN Nonce state, but this would severely limit Ack Vector's compressibility without providing much extra protection.)

Consider a half-connection from DCCP A to DCCP B. DCCP A SHOULD set ECN Nonces on its packets, and remember which packets had nonces, whenever DCCP B reports that it is ECN Capable. An ECN-capable endpoint MUST calculate and use the correct value for ECN Nonce Echo when sending acknowledgement options. An ECN-incapable endpoint, however, SHOULD treat the ECN Nonce Echo as always zero. When a sender detects an ECN Nonce Echo mismatch, it SHOULD behave as if the receiver had reported one or more packets as ECN-marked (instead of unmarked). It MAY take more punitive action, such as resetting the connection. The Reason for such DCCP-Reset packets SHOULD be set to "Aggression Penalty".

An ECN-incapable DCCP SHOULD ignore received ECN nonces and generate ECN nonces of zero. For instance, out of the two Ack Vector options, an ECN-incapable DCCP SHOULD generate Ack Vector [Nonce 0] (option 37) exclusively. (Again, the ECN Capable feature MUST be set to zero in this case.)

9.3. Other Aggression Penalties

The ECN Nonce provides one way for a DCCP sender to discover that a receiver is misbehaving. There may be other mechanisms, and a receiver or middlebox may also discover that a sender is misbehaving---sending more data than it should. In any of these cases, the entity that discovers the misbehavior MAY react by resetting the connection, with Reason set to "Aggression Penalty". A receiver that detects marginal (meaning possibly spurious) sender misbehavior MAY instead react with a Slow Receiver option, or by reporting some packets as ECN marked that were not, in fact, marked.

10. Multihoming and Mobility

DCCP provides primitive support for multihoming and mobility via a mechanism for transferring a connection endpoint from one address to another. The moving endpoint must negotiate mobility support beforehand, and both endpoints must share their Connection Nonces. When the moving endpoint gets a new address, it sends a DCCP-Move packet from that address to the stationary endpoint. The stationary endpoint then changes its connection state to use the new address.

DCCP's support for mobility is intended to solve only the simplest multihoming and mobility problems. For instance, DCCP has no support for simultaneous moves. Applications requiring more complex mobility semantics, or more stringent security guarantees, should use an existing solution like Mobile IP or [SB00].

10.1. Mobility Capable Feature

A DCCP uses the Mobility Capable feature to inform its partner that it would like to be able to change its address and/or port during the course of the connection.

Mobility Capable has feature number 5. The Mobility Capable feature located at DCCP A indicates whether or not A will accept a DCCP-Move packet sent by B. DCCP B sends a "Change(Mobility Capable, 1)" option to DCCP A to inform it that B might like to move later.

A Mobility Capable feature contains a single byte of data. Mobility is allowed if and only if this byte is nonzero. A DCCP MUST reject a DCCP-Move packet referring to a connection when Mobility Capable is 0; however, it MAY reject a valid DCCP-Move packet even when Mobility Capable is 1.

A new connection starts with Mobility Capable 0 (that is, mobility is not allowed) for both DCCPs.

10.2. Security

The DCCP mobility mechanism, like DCCP in general, does not provide cryptographic security guarantees. Nevertheless, mobile hosts must use valid sequence numbers and include valid Identifications in their DCCP-Move packets, providing protection against some classes of attackers. Specifically, an attacker cannot move a DCCP connection to a new address unless they know valid sequence numbers and how to generate valid Identifications. Even with the default MD5 Identification Regime, this means that an attacker must have snooped on every packet in the connection to get a reasonable probability of success, assuming that initial sequence numbers and Connection Nonces are chosen well (that is, randomly). Section 16 further describes DCCP security considerations.

10.3. Congestion Control State

Once an endpoint has transitioned to a new address, the connection is effectively a new connection in terms of its congestion control state: the accumulated information about congestion between the old endpoints no longer applies. Both DCCPs MUST initialize their congestion control state (windows, rates, and so forth) to that of a new connection---that is, they must "slow start"---unless they have high-quality information about actual network conditions between the two new endpoints. Normally, the only way to get this information would be by instrumenting a DCCP connection between the new addresses.

Similarly, the endpoints' configured MTUs (see 11) SHOULD be reinitialized, and PMTU discovery performed again, following an address change.

10.4. Loss During Transition

Several loss and delay events may affect the transition of a DCCP connection from one address to another. The DCCP-Move packet itself might be lost; the acknowledgement to that packet might be lost, leaving the mobile endpoint unsure of whether the transition has completed; and data from the old endpoint might continue to arrive at the receiver even after the transition.

To protect against lost DCCP-Move packets, the mobile host SHOULD retransmit a DCCP-Move packet if it does not receive an acknowledgement within a reasonable time period. Section 5.9 describes the mechanism used to protect against duplicate DCCP-Move packets.

A receiver MAY drop all data received from the old address/port pair once a DCCP-Move has successfully completed. Alternately, it MAY accept one Loss Window's worth of this data. Congestion and loss events on this data SHOULD NOT affect the new connection's congestion control state. The receiver MUST NOT accept data with the old address/port pair past one Loss Window, and SHOULD send DCCP-Resets in response to those packets.

During some transition period, acknowledgements from the receiver to the mobile host will contain information about packets sent both from the old address/port pair, and from the new address/port pair. The mobile DCCP MUST NOT let loss events on packets from the old address/port pair affect the new congestion control state.

11. Path MTU Discovery

A DCCP implementation SHOULD be capable of performing Path MTU (PMTU) discovery, as described in [RFC 1191]. The API to DCCP SHOULD allow this mechanism to be disabled in cases where IP fragmentation is preferred. The rest of this section assumes PMTU discovery has not been disabled.

A DCCP implementation MUST maintain its idea of the current PMTU for each active DCCP session. The PMTU SHOULD be initialized from the interface MTU that will be used to send packets.

To perform PMTU discovery, the DCCP sender sets the IP Don't Fragment (DF) bit. However, it is undersirable for MTU discovery to occur on the initial connection setup handshake, as the connection setup process may not be representative of packet sizes used during the connection, and performing MTU discovery on the initial handshake might unnecessarily delay connection establishment. Thus, DF SHOULD NOT be set on DCCP-Request and DCCP-Response packets. In addition DF SHOULD NOT be set on DCCP-Reset packets, although typically these would be small enough to not be a problem. On all other DCCP packets, DF SHOULD be set.

Any API to DCCP MUST allow the application to discover DCCP's current PMTU. DCCP applications SHOULD use the API to discover the PMTU, and SHOULD NOT send datagrams that are greater than the PMTU; the only exception to this is if the application disables PMTU discovery. If the application tries to send a packet bigger than the PMTU, the DCCP implementation MUST drop the packet and return an appropriate error.

As specified in [RFC 1191], when a router receives a packet with DF set that is larger than the PMTU, it sends an ICMP Destination Unreachable message to the source of the datagram with the Code indicating "fragmentation needed and DF set" (also known as a "Datagram Too Big" message). When a DCCP implementation receives a Datagram Too Big message, it decreases its PMTU to the Next-Hop MTU value given in the ICMP message. If the MTU given in the message is zero, the sender chooses a value for PMTU using the algorithm described in Section 7 of [RFC 1191]. If the MTU given in the message is greater than the current PMTU, the Datagram Too Big message is ignored, as described in [RFC 1191]. (We are aware that this may cause problems for DCCP endpoints behind certain firewalls.)

If the DCCP implementation has decreased the PMTU, and the sending application attempts to send a packet larger than the new MTU, the API MUST cause the send to fail returning an appropriate error to the application, and the application SHOULD then use the API to query the new value of the PMTU. When this occurs, it is possible that the kernel has some packets buffered for transmission that are smaller than the old PMTU, but larger than the new PMTU. The kernel MAY send these packets with the DF bit cleared, or it MAY discard these packets; it MUST NOT transmit these datagrams with the DF bit set.

DCCP currently provides no way to increase the PMTU once it has decreased.

A DCCP sender MAY optionally treat the reception of an ICMP Datagram Too Big message as an indication that the packet being reported was not lost due congestion, and so for the purposes of congestion control it MAY ignore the DCCP receiver's indication that this packet did not arrive. However, if this is done, then the DCCP sender MUST check the ECN bits of the IP header echoed in the ICMP message, and only perform this optimization if these ECN bits indicate that the packet did not experience congestion prior to reaching the router whose MTU it exceeded.

12. Middlebox Considerations

This section describes properties of DCCP that firewalls, network address translators, and other middleboxes must consider, including parts of the packet that middleboxes must not change.

The Service Name field in DCCP-Request packets provide information that may be useful for stateful middleboxes. With Service Name, a middlebox can tell what protocol a connection will use, without relying on port numbers. Middleboxes MAY disallow attempted connections with zero Service Names by sending a DCCP-Reset. Middleboxes SHOULD NOT modify the Service Name.

The Source and Destination Port fields are in the same packet locations as the corresponding fields in TCP and UDP, which may simplify some middlebox implementations.

Middleboxes MUST NOT modify DCCP packets' Sequence Number, Acknowledgement Number, and # NDP fields in order to add or remove packets from a packet stream. Any such modification would affect the endpoints' accounting of which packets have been lost, destroy the Identification mechanism, and confuse the congestion control mechanisms in use. Note that there is less need to modify DCCP's per-packet sequence numbers than TCP's per-byte sequence numbers; for example, a middlebox can change the contents of a packet without changing its sequence number. (In TCP, sequence number modification is required to support legacy protocols like FTP that carry variable-length addresses in the data stream. If such an application were deployed over DCCP, middleboxes would simply grow or shrink the relevant packets as necessary, without changing their sequence numbers.)

The exception to this rule is that middleboxes MAY reset connections in progress. Clearly this requires inserting a packet into one or both packet streams, as well as dropping all later packets on the connection.

This does not explicitly prevent one sequence number modification occasionally seen with TCP, namely proxies with "connection splicing" [SHHP00]. Such proxies intercept TCP connection attempts from a client, but may later "splice" data from an external server connection into that client connection via sequence number manipulations. Packets are not added to or removed from the spliced-in stream, reducing the sequence number issues somewhat. Nevertheless, DCCP, with its extensive end-to-end feature negotiation, is inherently unfriendly to the idea of connection splicing: the proxy would have to ensure that the server chose the same feature values that the proxy had previously negotiated with the client. Furthermore, Identification options would require special handling; and there may be other issues. We suggest that DCCP splicing, if implemented, should take place at the application level.

A middlebox that wants to trivially support the MD5 Identification Regime (Section 6.4.3) MUST NOT alter packets' Sequence Number, Type, and CCval fields, or the Connection Nonce feature values, which are included in the MD5 hash sent with Identification and Challenge options.

The contents of this section SHOULD NOT be interpreted as a wholesale endorsement of stateful middleboxes.

13. Abstract API

API issues for DCCP are discussed in another Internet-Draft, in progress.

14. Multiplexing Issues

In contrast to TCP, DCCP does not offer reliable ordered delivery. As a consequence, with DCCP there are no inherent performance penalties in layering functionality above DCCP to multiplex several sub-flows into a single DCCP connection.

However, this approach of multiplexing sub-flows above DCCP will not work in circumstances such as RTP where the RTP subflows require separate port numbers. In this case, if it is desired to share congestion control state among multiple DCCP flows that share the same source and destination addresses, the possibilities are to add DCCP-specific mechanisms to enable this, or to use a generic multiplexing facility like the Congestion Manager [RFC 3124] residing below the transport layer. For some DCCP flows, the ability to specify the congestion control mechanism might be critical, and for these flows the Congestion Manager will only be a viable tool if it allows DCCP to specify the congestion control mechanism used by the Congestion Manager for that flow. Thus, to allow the sharing of congestion control state among multiple DCCP flows, the alternatives seem to be to add DCCP-specific functionality to the Congestion Manager, or to add a similar layer below DCCP that is specific to DCCP. We defer issues of DCCP operating over a revised version of the Congestion Manager, or over a DCCP-specific module for the sharing of congestion control state, to later work.

15. DCCP and RTP

The real-time transport protocol, RTP [RFC 1889], is currently used (over UDP) by many of DCCP's target applications (for instance, streaming media). This section therefore discusses the relationship between DCCP and RTP, and in particular, the question of whether any changes in RTP are necessary or desirable when it is layered over DCCP instead of UDP. The main issue here is header size: a DCCP header is at least 4 bytes larger than a UDP header.

There are two potential sources of overhead in the RTP-over-DCCP combination, duplicated acknowledgement information and duplicated sequence numbers. We argue that together, these sources of overhead add just 4 bytes per packet relative to RTP-over-UDP, and that eliminating the redundancy would not reduce the overhead. However, particular CCIDs might make productive use of the space occupied by RTP's sequence number.

First, consider acknowledgements. The information on packet loss that RTP communicates via RTCP SR/RR packets is communicated by DCCP via acknowledgement options. Much of the information in an RTCP receiver report could be divined from DCCP acknowledgements, depending on the CCID in use. Acknowledgement options, such as Ack Vector, can be frequent and verbose, whereas RTCP reports are sent only rarely, with a minimum interval of 5 seconds between reports [RFC 1889].

However, not all CCIDs require such verbose acknowledgements. CCID 3 (TFRC) reports acknowledgements at a low rate---between 16 and 32 bytes of options (depending on ECN usage), sent once per round trip time. This is not an undue burden. Furthermore, the options are necessary to implement responsive congestion control, and we cannot report less frequently, although we might design alternative acknowledgement options that take fewer bytes. DCCP gives the application the trade off between small packet overhead and the precise feedback provided by Ack Vector.

While RTP receiver reports might be considered "redundant" in the presence of DCCP's more precise acknowledgements, they are sent so infrequently that it is not worth optimizing them away. Also, note that in the common case of a one-way data stream, acknowledgement packets contain no data, so acknowledgement header size (as distinct from congestion on the acknowledgement path) is not an issue.

We now consider sequence number redundancy on data packets. The embedded RTP header contains a 16-bit RTP sequence number. Most data packets will use the DCCP-Data type; DCCP-DataAck and DCCP-Ack packets need not usually be sent. The DCCP-Data header is 12 bytes long without options, including a 24-bit sequence number. This is 4 bytes more than a UDP header. Any options required on data packets would add further overhead, although many CCIDs (for instance, CCID 3 [TFRC]) don't require options on most data packets.

The DCCP sequence number cannot be inferred from the RTP sequence number since it increments on non-data packets as well as data packets. The RTP sequence number could be inferred from the DCCP sequence number, though; it might equal the DCCP sequence number minus the total number of non-data packets seen so far in the connection (as tracked by DCCP's # NDP header field).

Removing RTP's sequence number would not save any header space because of alignment issues. However, particular DCCP CCIDs might make use of the 16 bits occupied by the RTP sequence number. Therefore, particular DCCP CCIDs MAY provide optional CCID-specific features that store DCCP quantities in place of the embedded RTP sequence number. A conforming DCCP would write in the calculated RTP sequence number before passing the packet to RTP. (The DCCP checksum would use the DCCP quantity, not the RTP sequence number.)

Given RTP-over-DCCP's small overhead, however, implementors demanding tiny headers will probably prefer more comprehensive header compression to this ad-hoc compression technique.

16. Security Considerations

DCCP does not provide cryptographic security guarantees. Applications desiring hard security should use IPsec or end-to-end security of some kind.

Nevertheless, DCCP is intended to protect against some classes of attackers. Attackers cannot hijack a DCCP connection (close the connection unexpectedly, or cause attacker data to be accepted by an endpoint as if it came from the sender) unless they can guess valid sequence numbers. Thus, as long as endpoints choose initial sequence numbers well, a DCCP attacker must snoop on data packets to get any reasonable probability of success. The sequence number validity (Section 5.2), Identification (Section 6.4.3), and mobility

(Section 10) mechanisms provide this guarantee.

17. IANA Considerations

DCCP introduces six sets of numbers whose values should be allocated by IANA.

- 32-bit Service Names (Section 5.4; not exclusive to DCCP).
- 8-bit DCCP-Reset Reasons (Section 5.8).
- 8-bit DCCP Option Types (Section 6). The CCID-specific options 128 through 255 need not be allocated by IANA.
- 8-bit DCCP Feature Numbers (Section 6.3). The CCID-specific features 128 through 255 need not be allocated by IANA.
- 8-bit DCCP Congestion Control Identifiers (CCIDs) (Section 7).
- 16-bit Identification Regimes, for use with DCCP Identification and Challenge options (Section 6.4).

In addition, DCCP requires a Protocol Number to be added to the registry of Assigned Internet Protocol Numbers. Experimental implementors should use Protocol Number 33 for DCCP, but this number may change in future.

18. Design Motivation

In the section we attempt to capture some of the rationale behind specific details of DCCP design.

18.1. CSlen and Partial Checksumming

A great deal of discussion has taken place regarding the utility of allowing a DCCP sender to restrict the checksum so that it does not cover the complete packet.

Many of the applications that we envisage using DCCP are resilient to some degree of data loss, or they would typically have chosen a reliable transport. Some of these applications may also be resilient to data corruption---some audio payloads, for example. These resilient applications might prefer to receive corrupted data than to have DCCP drop a corrupted packet. This is particularly because of congestion control: DCCP cannot tell the difference between packets dropped due to corruption and packets dropped due to congestion, and so it must reduce the transmission rate accordingly. This response may cause the connection to receive less bandwidth than it is due; corruption in some networking technologies is independent of, or at least not always correlated to, congestion. Therefore, corrupted packets do not need to cause as strong a reduction in transmission rate as the congestion response would dictate (so long as the DCCP header and options are not corrupt).

Thus DCCP allows the checksum to cover all of the packet, just the DCCP header, or both the DCCP header and some number of bytes from the payload. If the application cannot tolerate any payload corruption, then the checksum SHOULD cover the whole packet. If the application would prefer to tolerate some corruption rather than have the packet dropped, then it can set the checksum to cover only part of the packet (but always the DCCP header). In addition, if the application wishes to decouple checksumming of the DCCP header from checksumming of the payload, it may do so by including the Payload Checksum option. This would allow payload corruption to cause DCCP to discard a corrupted payload, but still not mistake the corruption for network congestion.

Thus, from the application point of view, partial checksums seem to be a desirable feature. However, the usefulness of partial checksums depends on partially corrupted packets being delivered to the receiver. If the link-layer CRC always discards corrupted packets, then this will not happen, and so the usefulness of partial checksums would be restricted to corruption that occurred in routers and other places not covered by link CRCs. There does not appear to be consensus on how likely it is that future network links that suffer significant corruption will not cover the entire packet with a single strong CRC. DCCP makes it possible to tailor such links to the application, but it is difficult to predict if this will be compelling for future link technologies.

In addition, partial checksums do not co-exist well with IP-level authentication mechanisms such as IPsec AH, which cover the entire packet with a cryptographic hash. Thus, if cryptographic authentication mechanisms are required to co-exist with partial checksums, the authentication must be carried in the DCCP payload. A possible mode of usage would appear to be similar to that of Secure RTP. However, such "application-level" authentication does not protect the DCCP option negotiation and state machine from forged packets. An alternative would be to use IPsec ESP, and use encryption to protect the DCCP headers against attack, while using the DCCP header validity checks to authenticate that the header is from someone who possessed the correct key. However, while this is resistant to replay (due to the DCCP sequence number), it is not by itself resistant to some forms of man-in-the-middle attacks because the payload is not tightly coupled to the packet header. Thus an application-level authentication probably needs to be coupled with IPsec ESP or a similar mechanism to provide a reasonably complete security solution. The overhead of such a solution might be unacceptable for some applications that would otherwise wish to use partial checksums.

On balance, the authors believe 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. It remains to be seen whether they are useful in practice.

19. Thanks

There is a wealth of work in this area, including the Congestion Manager. We thank the staff and interns of ICIR and, formerly, ACIRI, the members of the End-to-End Research Group, and the members of the Transport Area Working Group for their feedback on DCCP. We also thank those who provided comments and suggestions via the DCCP BOF, Working Group, and mailing lists, including Damon Lanphear, Patrick McManus, Sara Karlberg, Kevin Lai, Youngsoo Choi, Dan Duchamp, Derek Fawcus, David Timothy Fleeman, John Loughney, Ghyslain Pelletier, Tom Phelan, Stanislav Shalunov, Yufei Wang, and Michael Welzl.

20. Normative References

[RFC 793] J. Postel, editor. Transmission Control Protocol. RFC 793.

[RFC 1191] J. C. Mogul and S. E. Deering. Path MTU Discovery. RFC 1191.

[RFC 2026] S. Bradner. The Internet Standards Process---Revision 3. RFC 2026.

[RFC 2119] S. Bradner. Key Words For Use in RFCs to Indicate Requirement Levels. RFC 2119.

[RFC 2460] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460.

[RFC 3168] K.K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168. September 2001.

21. Informative References

[CCID 2 PROFILE] S. Floyd and E. Kohler. Profile for DCCP Congestion Control ID 2: TCP-like Congestion Control. draft-ietf-dccp-ccid2-01.txt, work in progress, March 2003.

[CCID 3 PROFILE] S. Floyd, E. Kohler, and J. Padhye. Profile for DCCP Congestion Control ID 3: TFRC Congestion Control. draft-ietf-dccp-ccid3-01.txt, work in progress, March 2003.

[ECN NONCE] David Wetherall, David Ely, and Neil Spring. Robust ECN Signaling with Nonces. draft-ietf-tsvwg-tcp-nonce-04.txt, work in progress, October 2002.

[RFC 1889] Audio-Video Transport Working Group, H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 1889.

[RFC 1948] S. Bellovin. Defending Against Sequence Number Attacks. RFC 1948.

[RFC 2960] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960.

[RFC 3124] H. Balakrishnan and S. Seshan. The Congestion Manager. RFC 3124.

[SB00] Alex C. Snoeren and Hari Balakrishnan. An End-to-End Approach to Host Mobility. Proc. 6th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM '00), August 2000.

[SHHP00] Oliver Spatscheck, Jorgen S. Hansen, John H. Hartman, and Larry L. Peterson. Optimizing TCP Forwarder Performance. IEEE/ACM Transactions on Networking 8(2):146-157, April 2000.

[UDP-LITE] L-A. Larzon, M. Degermark, S. Pink, L-E. Jonsson (editor), and G. Fairhurst (editor). The UDP-Lite Protocol. draft-ietf-tsvwg-udp-lite-01.txt, work in progress, December 2002.

22. Authors' Addresses

Eddie Kohler <kohler@icir.org>
Mark Handley <mjh@icir.org>
Sally Floyd <floyd@icir.org>

ICSI Center for Internet Research
1947 Center Street, Suite 600
Berkeley, CA 94704 USA

Jitendra Padhye <padhye@microsoft.com>

Microsoft Research
One Microsoft Way
Redmond, WA 98052 USA