Internet Engineering Task Force
INTERNET-DRAFT
draft-ietf-dccp-spec-05.ps
Expires: April 2004

Eddie Kohler
UCLA/ICIR
Mark Handley
Sally Floyd
ICIR
Jitendra Padhye
Microsoft Research

27 October 2003

# Datagram Congestion Control Protocol (DCCP)

## Status of this Memo



## Copyright Notice



## 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, Internet telephony, and on-line games.

TO BE DELETED BY THE RFC EDITOR UPON PUBLICATION:

Changes since draft-ietf-dccp-spec-04.txt:

* Rearchitected feature negotiation (Junwen Lai).

* Added figures, and modified text, to the Overview section. Figures and text partly from Eric Rescorla.

* New synchronziation mechanism: DCCP-Sync.

* DCCP-Move: Add Mobility ID and remove Old Address and Old Port, because they wouldn't work through a NAT.

* The MD5 ID Regime is now number 1. (It is still the default.) ID Regime 0 is the Null Regime. Also switch the meaning of the ID Regime feature.

* Rename Drop States to Drop Codes, and renumber them.

* Ignored cannot contain more option data bytes than the offending option.

* Rename Service Name to Service Code (Gorry Fairhurst).

* Rename Cslen/Checksum Length to CsCov/Checksum Coverage and change its values by analogy with UDP-Lite.

* Be more specific about what Slow Receiver means.

* Allow a textual error message in DCCP-Reset.

* Mention new PMTUD, but this mention needs work.

* CCID 1: Specify when acks may be sent.

* Specify Request retransmission strategy.

* Other changes throughout.

Changes since draft-ietf-dccp-spec-03.txt:

* Specify how the Loss Window is arranged.

* Ignored can contain multiple bytes of option data.

* Refine the tables in Section 8.5.1, on Ack Vector Consistency.

* CC mechanisms must treat Data Dropped like ECN Marked unless otherwise specified.

* An MTU is mandatory (although PMTU is not), and CCIDs can affect the MTU.

* Clarifications in response to reviewer comments.

Changes since draft-ietf-dccp-spec-02.txt:

* Identification options include the Acknowledgement Number in their hash.

* Added an additional condition to accepting a packet with an invalid Sequence Number: the Acknowledgement Number must be valid, as well as the Identification options.

* Explicitly allow Connection Nonces to be negotiated in other ways than the Connection Nonce feature.

* Bad Moves are ignored, not reset, to avoid leaking information to attackers.

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 [NB: now Drop Code 1].

* 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.

# 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 semi-reliability, 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.

Different forms of conformant congestion control are appropriate for different applications, and a second motivation behind the design of DCCP is to allow applications to 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. Applications using this congestion control mechanism will respond quickly to changes in available bandwidth, but must be able to tolerate the abrupt changes in congestion window typical of TCP. A second alternative, TCP-Friendly Rate Control (TFRC), a form of equation-based congestion control, minimizes abrupt changes in the sending rate while maintaining longer-term fairness with TCP. TCP-like Congestion Control is appropriate for applications such as on-line games that want to make use of all the available bandwidth quickly, but can tolerate rapid reductions in rate without serious consequences. TFRC is more appropriate for applications such as streaming media, where rapid rate changes cause unacceptable UI glitches (audible pauses or clicks in the playout stream, for example). These applications would prefer to give up on rapid consumption of available bandwidth in favor of a steadier rate.

DCCP also allows unreliable traffic to use ECN safely. A UDP kernel API might not allow applications to set UDP packets as ECN-capable, since the API could not guarantee the application would properly detect or respond to congestion. DCCP kernel APIs will have no such issues, since DCCP itself implements congestion control.

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 [RFC 3124] 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. Conventions and Terminology

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 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].

All multi-byte numerical quantities in DCCP, such as Sequence Numbers and arguments to options, are transmitted in network byte order (most significant byte first).

We occasionally refer to the "left" and "right" sides of a bit field. "Left" means towards the most significant bit, and "right" means towards the least significant bit.

Reserved bitfields in DCCP packet headers MUST be ignored by receivers, and MUST be set to zero by senders, unless otherwise specified.

### 3.1. Robustness Principle

DCCP implementations should follow TCP's "general principle of robustness": be conservative in what you do, be liberal in what you accept from others.

### 3.2. Packet Types

DCCP has ten different packet types.

The DCCP-Request and DCCP-Response packets are used in connection initiation, and the DCCP-CloseReq, DCCP-Close, and DCCP-Reset packets are used in connection termination, as described in Section 4.1.

The other five packet types are as follows:

**DCCP-Data**
Used to transmit data. It carries no acknowledgement information.

**DCCP-Ack**
Used for pure acknowledgements.

**DCCP-DataAck**
Used for piggybacked data-plus-acknowledgements.

**DCCP-Move**
Supports multihoming and mobility.

**DCCP-Sync**
Used to resynchronize sequence numbers after a large burst of loss.

All of these packets except for DCCP-DataAck, DCCP-Move, and DCCP-Sync are shown in the example diagram below.

### 3.3. States

DCCP endpoints progress through different states during the course of a connection. The figure below shows the typical progress through these states for a client and server.

```
Client State:                                       Server State:
-------------                                       -------------
CLOSED                                                 LISTEN
REQUEST                    DCCP-Request ->
                        <- DCCP-Response               RESPOND
OPEN                       DCCP-Ack ->
                        <- DCCP-Data                   OPEN
                           DCCP-Ack ->
                        <- DCCP-CloseReq               CLOSEREQ
CLOSING                    DCCP-Close ->
                        <- DCCP-Reset                  CLOSED
TIME-WAIT
CLOSED
    The client and server's typical progress through states.
```

**CLOSED**
Represents a nonexistent connection.

**LISTEN**
Represents a server socket in the passive listening state. LISTEN and CLOSED are

not associated with any particular DCCP connection.

**REQUEST**
The client socket enters this state, from CLOSED, after sending a DCCP-Request packet to try to initiate a connection.

**RESPOND**
A server socket enters this state, from LISTEN, after receiving a DCCP-Request from a client.

**OPEN**
The central, data transfer portion of a DCCP connection. Client and server enter into this state from REQUEST and RESPOND, respectively. Sometimes we speak of SERVER-OPEN and CLIENT-OPEN states, corresponding to the server's OPEN state and the client's OPEN state.

**CLOSEREQ**
A server socket enters this state, from SERVER-OPEN, to signal that the connection is over, but the client must hold Time-Wait state.

**CLOSING**
Either server or client can enter this state to close the connection.

**TIME-WAIT**
A socket remains in this state for 2MSL after the connection has been torn down, to prevent mistakes due to the delivery of old packets.

## 3.4. Parts of a Connection

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.

These four subflows are grouped into two half-connections, illustrated as follows:

```
+--------+      A-to-B half-connection:                  +--------+
|        |    + - - - - - - - - - - - - - - - - - - - - +    |        |
|        |    |                 (1)                     |    |        |
|        |        data packets -->                           |        |
|        |    |                 (2)                     |    |        |
|        |                      <-- acknowledgements         |        |
|        |    + - - - - - - - - - - - - - - - - - - - - +    |        |
| DCCP A |                                                   | DCCP B |
|        |      B-to-A half-connection:                      |        |
|        |    + - - - - - - - - - - - - - - - - - - - - +    |        |
|        |    |                 (3)                     |    |        |
|        |                      <-- data packets             |        |
|        |    |                 (4)                     |    |        |
|        |      acknowledgements -->                         |        |
+--------+    + - - - - - - - - - - - - - - - - - - - - +    +--------+
```

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.

# 4. Overview

## 4.1. 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".

```
Client                                          Server
------                                          ------
DCCP-Request            ->
[Ports, service, features]
                        <-           DCCP-Response
                                [Features, cookie]
DCCP-Ack                ->
[Features, cookie]

               DCCP connection initiation.
```

In the DCCP-Request message, the client tells the server the ports it wants to communicate on and possibly the Service Code of the service it wants to talk to. The DCCP-Request message also starts feature negotiation, which, for pedagogical reasons, we will present separately in the next section.

In the DCCP-Response message, the server tells the client that it is willing to accept the connection and continues feature negotiation. In order to prevent SYN-flood style DOS attacks, DCCP incorporates a cookie exchange: The server can provide the client with a cookie that contains all the negotiation state. This cookie must be echoed by the client in the DCCP-Ack, thus removing the need for the server to keep state.

In the DCCP-Ack message, the client acknowledges the DCCP-Response and returns the cookie to permit the server to complete its side of the connection. This message may also include feature negotiation messages.

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. This preserves the invariant that every connection has one client and one server.

The server sends a DCCP-CloseReq packet to the client to ask it to close the connection with a DCCP-Close. The server sends DCCP-CloseReq, rather than DCCP-Close, when it wants the client to hold Time-Wait state for the connection. Only the server may generate a DCCP-CloseReq packet. This means that the client cannot force the server to maintain connection state after the connection is closed.

An endpoint sends a DCCP-Close packet to request that the other endpoint tear down the connection via DCCP-Reset. Every explicitly-terminated connection ends with a DCCP-Reset packet. The receiver of DCCP-Reset holds Time-Wait state for the connection. DCCP-Reset is sent in response to DCCP-Close during normal connection termination, or due to some inappropriate protocol event.

```
Client                                              Server
------                                              ------
                        <-            DCCP-CloseReq
DCCP-Close              ->
                        <-               DCCP-Reset

               DCCP connection termination.
```

DCCP shuts down both half-connections as a unit; it has no states analogous to TCP's FINWAIT and CLOSEWAIT states, where one TCP "half-connection" is closed and the other remains open. However, DCCP implementations 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 Code 1---for more information.

## 4.2. Congestion Control

Each half-connection is managed by a congestion control mechanism named by a single-byte congestion control identifier, or CCID. 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. The endpoints negotiate their CCIDs at connection setup; the CCIDs for the two half-connections need not be the same.

Section 7 introduces the currently allocated CCIDs, which are defined in separate profile documents.

### 4.2.1. CCID 2

CCID 2's congestion control is extremely similar to that of TCP. The sender maintains a congestion window and sends packets until that window is full. Packets are acknowledged by the receiver. Dropped packets and ECN [RFC 3168] are used to indicate congestion. The response to congestion is to halve the congestion window. One subtle diference between DCCP and TCP is that the acknowledgements in DCCP contain the sequence numbers of all received packets within a given window, not just the highest sequence number as in TCP's cumulative ackowledgement.

#### 4.2.2. CCID 3

CCID 3 is an equation-based form of congestion control which is intended to provide a smoother response to congestion than CCID 2. The sender maintains a "transmit rate". The receiver sends acknowledgement packets which also contain information about the receiver's estimate of packet loss. The sender uses this information to update its transmit rate. Although CCID 3 behaves somewhat differently from TCP in its short term congestion response, it is designed to operate fairly with TCP over the long term.

### 4.3. Features

In DCCP, feature negotiation is performed by attaching options to other DCCP packets. Thus feature negotiation can be piggybacked on any other DCCP message. This allows feature negotiation during connection initiation as well as feature renegotiation during data flow.

DCCP features are one-sided. Thus, it's possible to have a different congestion control regime for data sent from client to server than from server to client. The endpoint in charge of a particular feature is called its feature location; the other endpoint is called the feature remote. Feature negotiation is done with the Change L, Confirm L, Change R, and Confirm R options, with the "L" options sent by the feature location, and "R" options sent by the feature remote.

A Change R message says to the peer "change this option setting on your side". The peer responds with a Confirm L, meaning "I've changed it". Some sample exchanges follow:

```
Client                                            Server
------                                            ------
Change R(CCID, 2)       ->
                        <-        Confirm L(CCID, 2)
           * agreement that (CCID,Server) = 2 *
```

In this exchange, the peers agree to set the server's CCID to 2.

```
Client                                            Server
------                                            ------
Change R(CCID, 3 4)      ->
                         <-  Confirm L(CCID, 4, 4 2)
           * agreement that (CCID,Server) = 4 *
```

In this exchange, the client requests CCID value 3 or 4 for the server's CCID, with 3 preferred. Note that the client can offer multiple values. The server chooses 4, giving its preference list of "4 2".

If a party wants to change one of his own options, he issues a "Change L", as shown below.

```
Client                                            Server
------                                            ------
                         <-      Change L(CCID, 3 2)
Confirm R(CCID, 3, 3 2)  ->
           * agreement that (CCID, Server) = 3 *
```

In this example, the server requests CCID value 3 or 2 for the server's CCID, with 3 preferred, and the client agrees.

Retransmissions make feature negotiation reliable. Section 6.4 describes these options further.

## 4.4. Example Connection

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

```
Client                                       Server
------                                       ------
(1) DCCP-Request        ->
                        <-        (2) DCCP-Response
(3) DCCP-Ack            ->
(5) DCCP-Data           ->
                        <-             (5) DCCP-Ack
                        <-            (5) DCCP-Data
(5) DCCP-Ack            ->
                        <-        (6) DCCP-CloseReq
(7) DCCP-Close          ->
                        <-           (8) DCCP-Reset

             Typical DCCP Connection.
```

(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. In DCCP, unlike TCP, Resets are part of normal connection termination; see Section 5.9.

(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 Time-Wait state at connection closedown.

## 4.5. 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.

### 4.5.1. 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 R option asking the server to use CCID 2 for the server's data packets, and a Change L 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 L option indicating that the server agrees to use CCID 2 for its data packets, and a Confirm R 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 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 Change R(Ack Ratio) feature options specifying the number of server data packets to be covered by an Ack packet from the client.

Each DCCP-Data 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 L options answering any Ack Ratio requests from the server.

The 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 must occasionally acknowledge the client's acknowledgements, so the client can clean its acknowledgement state. It can do so by sending separate DCCP-Acks as allowed by CCID 2, or by piggybacking acknowledgement information on its data packets with the DCCP-DataAck packet type. The acknowledgement information may contain detailed Ack Vectors, like the client's acknowledgements; but if the client is sending nothing but acknowledgements, the server's acks-of-acks can be more lightweight. See Section 8.1 for more information.

Like the server's DCCP-Data packets, the server's DCCP-DataAck and DCCP-Ack packets are sent as ECN-Capable.

(7) 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.

(8) Because DCCP-Ack packets use sequence numbers, the server has direct information about the fraction of loss or marked DCCP-Ack packets. [CCID 2 PROFILE] defines how the server modifies the client's Ack Ratio in response to any congestion on the acknowledgement stream.

(9) 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.

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

#### 4.5.2. 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           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data Offset  | CCVal | CsCov |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Type  |X|# NDP|              Sequence Number                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

**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.

**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.

**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. If the relevant CCID does not specify its value, it MUST be set to zero.

**Checksum Coverage (CsCov): 4 bits**
The Checksum Coverage field specifies what parts of the packet are covered by the Checksum field, as follows:

**CsCov = 0**
Checksum covers the DCCP header, DCCP options, network-layer pseudoheader (described below), and the entire DCCP payload, possibly padded on the right with zeros to an even number of bytes.

**CsCov = 1-15**

Checksum covers the DCCP header, DCCP options, network-layer pseudoheader, and the initial (CsCov-1)*4 bytes of the DCCP payload.

Thus, if CsCov is 1, none of the DCCP payload is protected by the header checksum. The value (CsCov-1)*4 MUST be less than or equal to the length of the DCCP payload. Packets with invalid CsCov values MUST be ignored; in particular, their options MUST NOT be processed. The meanings of values other than 0 and 1 should be considered experimental.

Values other than 0 specify that corruption is acceptable in some or all of the DCCP packet's payload. In fact, DCCP cannot even detect corruption in areas not covered by the header checksum, unless the Payload Checksum option is used (Section 8.8). Applications should not make any assumptions about the correctness of received data not covered by the checksum, and should if necessary introduce their own appropriate validity checks.

A DCCP application interface should let sending applications suggest a value for CsCov for sent packets, defaulting to 0 (full coverage). It should also let receiving applications refuse delivery of packets with checksum coverage less than a value provided by the application; by default, only packets with fully-covered payloads should be accepted. Lower layers that support partial error detection MAY use the Checksum Coverage field as a hint of where errors do not need to be detected. Lower layers MUST use a strong error detection mechanism to detect at least errors that occur in the sensitive part of the packet, and discard damaged packets. The sensitive part consists of the bytes between the first byte of the IP header and the last byte identified by Checksum Coverage. For more details on application and lower-layer interface issues relating to partial checksumming, see [UDP-LITE], from which this text was summarized.

See Appendix B.1 for further motivation of partial checksums and discussion of partial checksumming issues. Partial checksums introduce some security considerations, which are described in Section 16.2. DCCP partial checksumming 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 Coverage 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 header checksums MUST be ignored. In particular, their options MUST NOT be processed.

**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.

**9** DCCP-Sync packet.

**10-15**
Reserved.

**Extended Sequence Numbers (X): 1 bit**

This bit is set to one to indicate the use of an extended generic header with 48-bit Sequence and Acknowledgement Numbers. The format described in the section has X set to zero. Section 5.3 describes the extended generic header.

**Number of Non-Data Packets (# NDP): 3 bits**

DCCP sets this field to the number of non-data packets it has sent so far on its sequence, modulo 8 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.

# NDP can overflow, causing ambiguities. For example, if 8 packets are dropped in a row but # NDP does not change, the receiver will not be able to tell whether or not any of the lost packets contained data. Thus, applications SHOULD NOT depend on the availability of unambiguous # NDP information. DCCP itself uses # NDP only as a hint of when a connection has left unidirectional mode; potential ambiguities are not harmful there.

**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 and half-open connections; see Section 5.2 on sequence number validity.

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.

If the header's X bit equals one, the Sequence Number field extends for another 24 bits for a total of 48. Very-high-rate connections SHOULD use these extended 48-bit sequence numbers to protect against wrapped sequence numbers; see Section 5.3.

Many packet types also carry an Acknowledgement Number in the four bytes following the generic header. Its format is as follows:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Reserved    |            Acknowledgement Number             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

**Acknowledgement Number: 24 bits**

The Acknowledgement Number field acknowledges the *greatest* valid sequence number received so far on this connection. ("Greatest" is, of course, measured in circular sequence space.) Acknowledgement numbers make no attempt to provide precise information about which packets have arrived; options such as the Ack Vector do this.

The Acknowledgement Number MUST correspond to a "received" packet, where a packet is classified as "received" if and only if its options were processed by the receiving DCCP. (This means, for example, that received packets must be both header-checksum-valid and sequence-valid.) Even "received" packets may have their payloads dropped, due to receive buffer overflow or payload corruption, for instance. The HC-Receiver will send Data Dropped options when this happens (see Section 8.7); the HC-Sender will reduce its sending rate or congestion window as appropriate. This issue is discussed further in Sections 8.5 and 8.7.

If the header's X bit equals one, the Acknowledgement Number field extends for another 24 bits for a total of 48. Again, see Section 5.3.

**Reserved: 8 bits**

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

## 5.2. Sequence Number Synchronization

DCCP implementations must react to packets that are not intended for the current connection. This can happen if the network delivers an old packet, if an attacker attempts to hijack a connection, during the cleanup of a half-open connection, or for other reasons. DCCP, like TCP, uses sequence number checks and Reset packets to defend against these packets. Every DCCP packet sent uses a new sequence number, however; thus, given large enough bursts of loss, a connection's endpoints might get out of sync relative to any window, requiring a mechanism to restore synchronization. This section describes the algorithms that determine when DCCP packets are intended for the current connection, and the actions taken on unintended packets.

### 5.2.1. Variables

DCCP sequence number synchronization depends on the following variables, which are maintained by each endpoint.

GSS The Greatest Sequence Number Sent by this endpoint so far. ("Greatest" is of course measured in circular sequence space.)

GSR The Greatest Sequence Number Received from the other endpoint so far.

GAR (Optional) The Greatest Acknowledgement Number Received from the other endpoint so far.

Some other variables are derived from these primitives.

SWL and SWR
(Sequence Number Window Left and Right) The two endpoints of the window within which Sequence Numbers are appropriate.

AWL and AWR
(Acknowledgement Number Window Left and Right) The two endpoints of the window within which Acknowledgement Numbers are appropriate.

### 5.2.2. Appropriate Sequence Numbers

A sequence number S is appropriate iff SWL <= S <= SWR in circular sequence space. This resembles TCP's receive window. However, in DCCP, 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, and a packet beyond the window is not necessarily a hard error. DCCP-Sync packets help in this situation.

DCCP A sets SWL and SWR to a loss window of W consecutive sequence numbers containing GSR. ("Consecutive", like "greatest", is measured in circular sequence space.) One-third of the loss window, rounded down, is placed at and before GSR, with two-thirds after GSR. Sequence numbers outside this loss window are inappropriate.

```
 inapprop. |     appropriate Sequence Numbers     | inapprop.
<---------*|*==========*=====================*|*--------->
      GSR -|GSR + 1 -   GSR                GSR +|GSR + 1 +
 floor(W/3)|floor(W/3)                  ceil(2W/3)|ceil(2W/3)
           = SWL                           = SWR
```

During connection startup, DCCP A MUST adjust SWL so that it is not less than DCCP B's initial sequence number.

DCCP B informs DCCP A of W, the loss window width DCCP A should use, via the Loss Window feature (Section 6.10). W defaults to 1000, but a proper value should reflect how many packets the sender expects to be in flight. Only the sender can anticipate this number. 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. One good guideline is to set it to about 3 or 4 times the maximum number of packets the sender expects to send in a round-trip time. This value may not be available at connection initiation, when the round-trip time is unknown, but the sender can always send updates as the connection progresses.

### 5.2.3. Appropriate Acknowledgement Numbers

The Acknowledgement Number on a packet from DCCP B is appropriate iff it lies within the window [AWL, AWR], where AWR = GSS, and the window is W' packets wide. W' is the value of DCCP A's Loss Window feature, which it defined in its role as HC-Sender for the other half-connection.

```
 inapprop. | appropriate Acknowledgement Numbers | inapprop.
<---------*|*==================================*|*---------->
   GSS - W'|GSS - W' + 1                      GSS|GSS + 1
           = AWL                           = AWR
```

During connection startup, DCCP A MUST adjust AWL so that it is not less than its initial sequence number.

### 5.2.4. Sequence-Validity By State

A packet is called *sequence-valid* when its sequence numbers indicate that it is intended for the current connection. The rules for sequence-validity depend on the state of the connection. The baseline rules for sequence-validity are as follows:

**CLOSED and LISTEN states**

All packets are sequence-valid (but most packet types will cause a Reset to be generated by later validity checks).

**REQUEST state**

A packet is sequence-valid if and only if it has an appropriate Acknowledgement Number.

**All other states**

(1) DCCP-Data packets are sequence-valid if and only if their Sequence Numbers are appropriate.

(2) DCCP-Sync and DCCP-Reset packets are sequence-valid if and only if their Acknowledgement Numbers are appropriate.

(3) The sequence-validity of DCCP-Move packets is discussed in Section 5.10.

(4) All other packets are sequence-valid if and only if both their Sequence and Acknowledgement Numbers are appropriate.

DCCP implementations MAY implement additional checks to protect against packets that have valid sequence numbers, but are not part of this connection. The additional checks provide an incremental security advantage at a moderate complexity cost.

- DCCP-Reset packets may not have valid Sequence Numbers because they might be generated by a closed connection in response to DCCP-Data packets, which have no Acknowledgement Number. However, DCCP implementations MUST supply a valid Sequence Number when one is available (either from connection information or the Acknowledgement Number), and use Sequence Number 0 otherwise. Thus, valid DCCP-Reset packets fall into two categories: Either they contain an appropriate Sequence Number, or they have Sequence Number 0 and their Acknowledgement Number corresponds to a DCCP-Request or DCCP-Data packet. Implementations that check this invariant MUST ignore DCCP-Resets that don't fit. (Do not, for example, send a DCCP-Sync in response to such a Reset.)
- DCCP implementations transition to CLOSED state after sending a DCCP-Reset packet, and will not send further non-Reset packets on that connection. Therefore, valid DCCP-

Reset packets have Sequence Numbers greater than GSR (except for those with Sequence Number 0, as mentioned above), and Acknowledgement Numbers greater than or equal to GAR. Again, implementations that check this invariant MUST ignore DCCP-Resets that don't fit.

- Implementations that can detect duplicate sequence numbers within the current Loss Window should ignore duplicate packets. (Of course, sequence number space can wrap; this refers to packets whose sequence numbers have recently been seen.)
- DCCP-Sync packets with Sequence Number less than GSR, or with Acknowledgement Number less than GAR, are stale and MUST be ignored when detected.

Implementing these checks should not cause interoperability problems, but augmenting the list with additional ad-hoc checks is NOT RECOMMENDED.

### 5.2.5. Handling Sequence-Invalid Packets

Sequence-invalid DCCP-Move, DCCP-Reset, and DCCP-Sync packets MUST be ignored.

Otherwise, on receiving a sequence-invalid packet, a DCCP endpoint (say DCCP A) MUST reply with a DCCP-Sync packet, as allowed by the congestion control mechanism in use. This packet MUST acknowledge the packet's Sequence Number (not GSR!). Any DCCP-Sync MUST use a new Sequence Number, and thus will increase GSS; GSR will not change, however, since the packet was sequence-invalid. DCCP A MUST NOT otherwise process sequence-invalid packets.

On receiving the DCCP-Sync, DCCP B will update its GSR variable and reply with a DCCP-Sync of its own. When DCCP A receives this DCCP-Sync, which acknowledges its DCCP-Sync (and is therefore sequence-valid), it will update its GSR variable, thus getting the endpoints back into sync. Alternatively, if the connection was half-open, DCCP B will send a Reset.

To protect itself against denial-of-service attacks (where an attacker sends purposefully invalid packets, thereby forcing the receiver to send DCCP-Syncs), a DCCP implementation MAY ignore packets with inappropriate Sequence Numbers if the connection is still active. By "ignore", we mean that the packet is discarded without sending a DCCP-Sync. A connection is "active" when appropriate Sequence Numbers have been recently received; "recently" might mean within the last second or the last RTT, whichever is shorter.

Similarly, a DCCP MAY rate-limit the DCCP-Syncs sent in response to sequence-invalid packets.

### 5.2.6. Examples

In this first example, DCCP A and DCCP B recover from a large burst of loss that runs DCCP A's sequence numbers out of DCCP B's appropriate sequence number window.

```
                  Recovery from Burst of Loss
DCCP A                                                  DCCP B
(GSS=1,GSR=10)                                          (GSS=10,GSR=1)
           --->   DCCP-Data(seq 2)     XXX
                      ...
           --->   DCCP-Data(seq 100)   XXX
           --->   DCCP-Data(seq 101)          --->  ???
                                                    seqno out of range;
                                                    send Sync
   OK      <---   DCCP-Sync(seq 11, ack 101)   <---
                                                    (GSS=11,GSR=1)
           --->   DCCP-Sync(seq 102, ack 11)   --->   OK
(GSS=102,GSR=11)                                    (GSS=11,GSR=102)
```

In this example, a DCCP connection recovers from a simple attack. The attacker cannot guess sequence numbers. (DCCP is not robust to attackers who can guess sequence numbers.)

```
                      Recovery from Attack
DCCP A                                                  DCCP B
(GSS=1,GSR=10)                                          (GSS=10,GSR=1)
             *ATTACKER*  --->  DCCP-Data(seq 10^6)  --->  ???
                                                    seqno out of range;
                                                    send Sync
   ???     <---   DCCP-Sync(seq 11, ack 10^6)   <---
ackno out of range; ignore
(GSS=1,GSR=10)                                          (GSS=11,GSR=1)
```

The final example demonstrates recovery from a half-open connection.

```
                 Recovery from a Half-Open Connection
DCCP A                                                  DCCP B
(GSS=1,GSR=10)                                          (GSS=10,GSR=1)
(Crash)
CLOSED                                                     OPEN
REQUEST    --->   DCCP-Request(seq 400)       --->   ???
!!         <---   DCCP-Sync(seq 11, ack 400)  <---   OPEN
REQUEST    --->   DCCP-Reset(seq 401, ack 11) --->   (Abort)
REQUEST                                CLOSED
REQUEST    --->   DCCP-Request(seq 402)       --->   ...
```

### 5.3. Extended Sequence Numbers

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. DCCP's current congestion control mechanisms are designed for congestion windows (or equivalents) of at most a few hundred thousand packets, leaving at least 32 RTTs before 24-bit sequence numbers wrap. However, very-high rate connections SHOULD use extended sequence numbers to gain more protection.

DCCP extended sequence numbers are activated when the header's X bit is set to one. This extends the Sequence Number and Acknowledgement Number fields by an additional 24 bits, for a total of 48 bits. A flow of 1500-byte DCCP packets would have to send more than 28 petabits per second to overflow 48-bit sequence numbers within the 2-minute maximum segment lifetime. The 48-bit numbers are stored in network order, with most significant bit first.

```
  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           |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |  Data Offset  | CCVal | CsCov |           Checksum            |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 | Type  |1|# NDP|          Sequence Number (high bits)          |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |            Sequence Number (low bits)         |  Reserved   |T|
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

All packet types except for DCCP-Data and DCCP-Request will follow this generic header with an extended Acknowledgement Number:

```
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |   Reserved    |       Acknowledgement Number (high bits)      |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |       Acknowledgement Number (low bits)       |   Reserved    |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Once an endpoint has sent any packet with 48-bit sequence numbers (X=1), it MUST send all succeeding packets with 48-bit sequence numbers. Furthermore, once an endpoint has received any packet with 48-bit sequence numbers, it MUST either send all succeeding packets with 48-bit sequence numbers, or reset the connection with Reason set to "Extended Sequence Numbers" (15).

Clients SHOULD decide whether to use extended sequence numbers before sending their DCCP-Requests. That is, connections SHOULD NOT transition from 24-bit to 48-bit sequence numbers; they SHOULD contain only 24-bit sequence numbers, or only 48-bit sequence numbers. The Transition bit (T) supports transitioning to extended sequence numbers during an active connection, however, in case this proves necessary; see below.

Extended sequence numbers are treated simply as longer sequence numbers. For instance, the sequence-validity mechanisms work the same way whether or not sequence numbers are extended. Care is required when comparing a 24-bit sequence number with an 48-bit sequence number; see below.

Extended sequence numbers improve security against attackers by making it harder to guess a valid sequence number, as well as protecting against benign wrapping.

### 5.3.1. Transitioning to Extended Sequence Numbers

The Transition bit (T) following the extended Sequence Number field makes it possible to transition to 48-bit sequence numbers in the middle of a connection. T is set to one only during such a transition. When DCCP A switches to 48-bit sequence numbers, it MUST set the T bit to one on all of its packets for some period. This period SHOULD last on the order of a few round trip times, or until DCCP A receives an acknowledgement from DCCP B proving that one of its 48-bit-sequence-number packets has been received, whichever comes later.

Each DCCP MUST choose its first 48-bit sequence number to have its lower 24 bits equal the 24-bit sequence number it expected to send (GSS+1). If DCCP A sends an extended packet containing an Acknowledgement Number before DCCP B sends it a 48-bit Sequence Number, DCCP A may send any value for the upper 24 bits of that Acknowledgement Number, but the lower 24 bits MUST equal the expected 24-bit Acknowledgement Number (GSR). Furthermore, DCCP A MUST leave GSR as a 24-bit number until receiving an extended packet from DCCP B. If DCCP B transitions to extended sequence numbers because it receives a valid packet with extended sequence numbers, it MAY set the upper 24 bits of its extended sequence number based on the upper 24 bits of the received Acknowledgement Number, but it can also choose a different upper 24 bits.

Switching to 48-bit sequence numbers in the middle of a connection raises the issue of comparing a 24-bit sequence number with a 48-bit sequence number. (This may also occur if the network delivers a packet from an old connection, or given a malicious attacker.) Let P be the packet sequence number received from DCCP B, and E be the sequence number DCCP A expects. During sequence-validity computations, for example, P might be the packet's Acknowledgement Number and E might be AWL, the left edge of the appropriate acknowledgement number window. Then DCCP A should perform the comparison as follows.

- If P and E are both 24 bits, compare them modulo 2^24.
- If P and E are both 48 bits, the packet's Transition bit is set, and the last packet sent by DCCP A had its Transition bit set, then compare P and E modulo 2^24. This covers the case where both endpoints transitioned simultaneously, so P and E's upper 24 bits might disagree.
- Otherwise, if P and E are both 48 bits, compare them modulo 2^48.
- If P is 48 bits but E is 24, the remote DCCP may want to transition to extended sequence numbers. If the packet's Transition bit is not set, the packet is definitely sequence-invalid; otherwise, compare P with E modulo 2^24. If the packet proves sequence-valid, then it is OK; transition to extended sequence numbers, and set E according to the full 48 bits of P. If the packet does not prove sequence-valid, send an (extended) DCCP-Sync as required (with T set to one), but do not yet transition to extended sequence numbers.
- If P is 24 bits but E is 48, there may have been benign packet reordering. The correct action depends on whether the last packet seen from the remote DCCP had the Transition bit set.
  - If Transition was not set, then the packet is sequence-invalid; send an (extended) DCCP-Sync as required.
  - If Transition was set, extend P to a 48-bit value P'. First, let EH equal the upper 24 bits of E, and EL equal the lower 24 bits of E. Then:

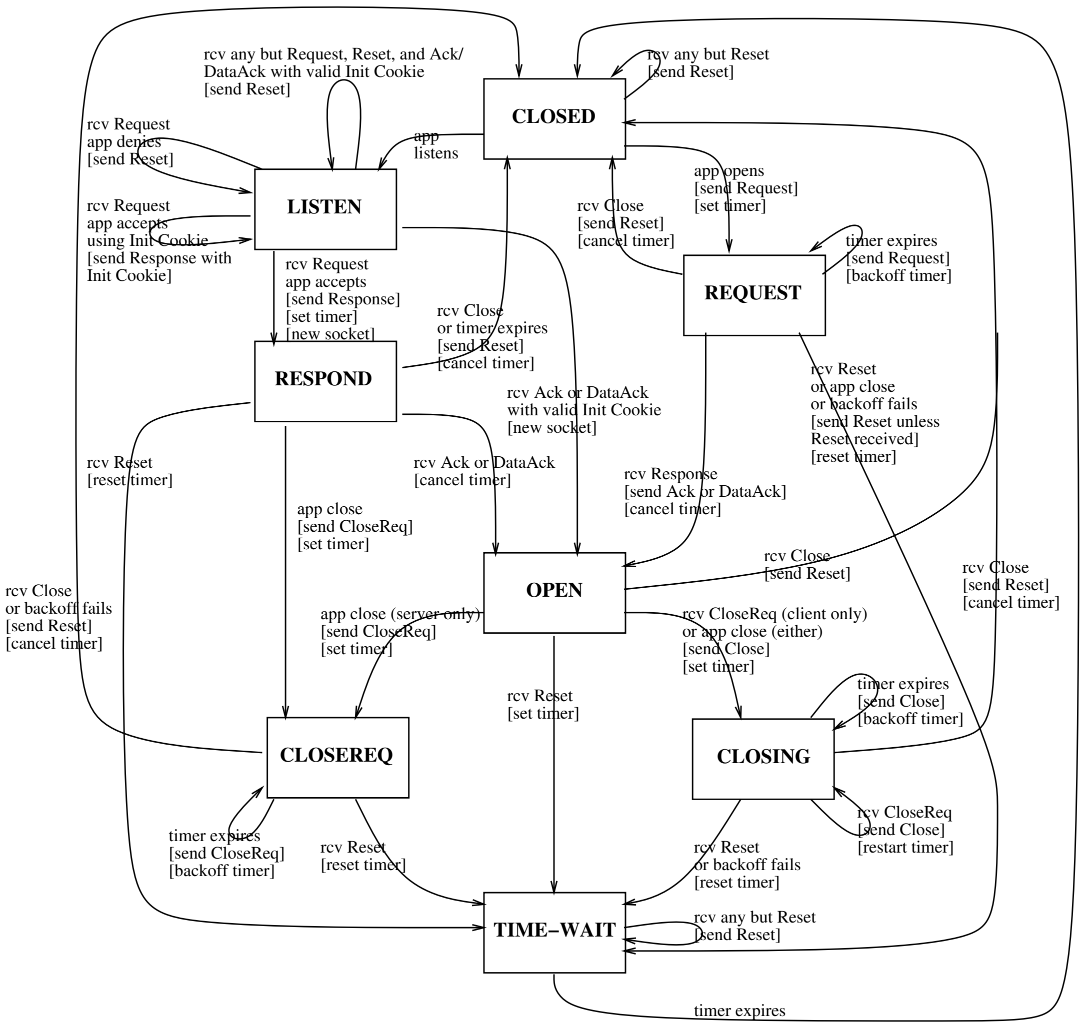
    ```
    If  EL > P,  set  P' = (EH << 24) | P.
    Otherwise,   set  P' = (((EH - 1) mod 2^24) << 24) | P.
    ```

    If the packet proves sequence-valid when comparing with P' modulo 2^48, then it is OK; the packet was reordered from before the transition. If it does not, send an (extended) DCCP-Sync (with T set to one) as required.

DCCP implementations can, of course, avoid most of this complexity by disallowing transitions to extended sequence numbers (and by resetting the connection when the other endpoint attempts such a transition). Connections that use 48-bit sequence numbers throughout, starting with the DCCP-Request, MUST have T set to zero on all their packets.

## 5.4. 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 sequence-valid packets with correct header checksums. For example, receiving a Reset with a bad Acknowledgement Number MUST NOT cause DCCP to transition to the TIME-WAIT state. DCCP implementations SHOULD send Acks as described above in response to sequence-invalid packets.

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 respond with a (possibly 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 GSR, the greatest valid sequence number seen so far).

| State | Request | Response | Data/Ack/ DataAck/ Move | CloseReq | Close | Reset/ Sync |
|---|---|---|---|---|---|---|
| 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 |
| SERVER-OPEN | -/Rst | Rst | OK | Rst | OK | OK |
| CLIENT-OPEN | Rst | -/Rst | OK | OK | OK | OK |
| CLOSEREQ | -/Rst | Rst | OK | Rst | OK | OK |
| CLOSING | Rst | -/Rst | OK | OK | OK | OK |
| TIME-WAIT | Rst | Rst | Rst | Rst | Rst | OK |

Again, we note that the table only applies to valid packets. Sequence-invalid packets SHOULD be treated as described above.

A DCCP endpoint that implements the Init Cookie option (Section 6.6) may change the Reset action marked (1). Init Cookie lets the server package all state for a requested connection into an option that the client will echo. A server with Init Cookie need not implement the RESPOND state. Instead, it may reply to each DCCP-Request packet with a DCCP-Response containing an Init Cookie. When a DCCP-Data, Ack, or DataAck packet carrying a valid Init Cookie arrives from the client, the server will move directly from LISTEN to OPEN. Like TCP SYN cookies [SYNCOOKIES], Init Cookies let servers avoid keeping any state for clients whose addresses have not been verified.

A DCCP endpoint in the CLOSED or LISTEN state may not have a proper sequence number available to send a Reset. In these cases, it MUST set the Reset's Sequence Number to zero. Resets sent in the CLOSED, LISTEN, and TIME-WAIT states SHOULD use Reset Reason "No Connection"; other Resets SHOULD use Reason "Invalid Packet". A DCCP MAY send Resets not listed in the diagram if it detects an inconsistency---for example, if it receives two DCCP packets with the same sequence number, but different packet types.

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 CLOSEREQ state. (The server is the DCCP endpoint that began in the LISTEN state.) Similarly, only the client must transition to CLOSE after receiving a CloseReq packet.

## 5.5. 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 or 16 bytes)             /
/                   with Type=0 (DCCP-Request)                  /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Service Code                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                  /    [padding]    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              data                             |
|                               ...                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

**Service Code: 32 bits**
The Service Code field describes the service to which the sender is trying to connect. Service Codes 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 Codes, stateful middleboxes, such as firewalls, can identify the application running on a nonstandard port (assuming the DCCP header has not been encrypted). A Service Code 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 Code. Connection requests MUST fail if the Destination Port on the receiver has a different Service Code from that given in the packet, and both Service Codes are nonzero. In this case, the receiver will respond with a DCCP-Reset packet (with Reason set to "Bad Service Code"). A server or stateful middlebox MAY also send a "Bad Service Code" DCCP-Reset in response to packets whose Service Code is considered unsuitable.

**Options**
DCCP-Request packets will usually include a "Change R(Connection Nonce)" option, to inform the server of the client's connection nonce; see Section 6.5.

The client MAY send new DCCP-Request packets if no response is received after some timeout. The retransmission strategy SHOULD be similar to that for retransmitting TCP SYNs; for instance, a first timeout on the order of a second, with an exponential backoff timer. Each retransmission MUST increment the Sequence Number, and possibly # NDP, by one.

A client MAY decide to give up after some number of DCCP-Requests. If so, it SHOULD 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 "Aborted".

## 5.6. 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 or 16 bytes)             /
 /                  with Type=1 (DCCP-Response)                  /
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |   Reserved    |             Acknowledgement Number            |
(+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+)Iff
(|       Acknowledgement Number (low bits)       |   Reserved    |)X=1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                     Options                   /   [padding]   |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                             data                              |
 |                              ...                              |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

**Acknowledgement Number: 24 bits**
In the case of a DCCP-Response packet, the Acknowledgement Number field will equal the sequence number from the corresponding DCCP-Request.

**Options**
The Data Dropped and Init Cookie options are particularly useful for DCCP-Response packets (Sections 8.7 and 6.6). In addition, DCCP-Response, or early DCCP-Data or DCCP-Ack packets, may include "Confirm L(Connection Nonce)" and "Change R(Connection Nonce)" options, to negotiate connection nonces (Section 6.5), 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 Code", when the DCCP-Request's Service Code did not correspond to the service code 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. (Note that the "retransmitted" DCCP-Request will have, at least, a different sequence number from the "original" DCCP-Request; the receiver can thus distinguish true retransmissions from network duplicates.) 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 while the server is in the RESPOND state MUST elicit a new DCCP-Response. Each new DCCP-Response MUST increment the responder's Sequence Number, and possibly # NDP, by one.

The responder SHOULD NOT accept any data accompanying a retransmitted DCCP-Request. In particular, the DCCP-Response sent in reply to a retransmitted DCCP-Request with data SHOULD contain a Data Dropped option, in which the retransmitted DCCP-Request is reported as "data dropped due to protocol constraints" (Drop Code 0). The original DCCP-Request SHOULD also be reported in the Data Dropped option, either in a Normal Block (if the responder accepted the data, or there was no data), or in a Drop Code 0 Drop Block (if the responder refused the data the first time as well).

## 5.7. DCCP-Data, DCCP-Ack, and DCCP-DataAck Packet Formats

The payload of a DCCP connection is sent in DCCP-Data and DCCP-DataAck packets, and 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 or 16 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 or 16 bytes)             /
 /                    with Type=3 (DCCP-Ack)                     /
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |   Reserved    |           Acknowledgement Number              |
(+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+)Iff
(|       Acknowledgement Number (low bits)       |   Reserved    |)X=1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                     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 or 16 bytes)             /
 /                  with Type=4 (DCCP-DataAck)                   /
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |   Reserved    |           Acknowledgement Number              |
(+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+)Iff
(|       Acknowledgement Number (low bits)       |   Reserved    |)X=1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                     Options                   /   [padding]   |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                             data                              |
 |                              ...                              |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

A DCCP-Data or DCCP-DataAck packet may contain no data bytes if the application sends a zero-length datagram.

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 congestion state of the A-to-B CCID will not allow data to be sent.

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

Section 8, below, describes acknowledgements in DCCP.

## 5.8. DCCP-CloseReq and DCCP-Close Packet Format

The DCCP-CloseReq and DCCP-Close packets have the same format except for Type. 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 Time-Wait state. The receiver of a valid DCCP-CloseReq packet SHOULD respond with a DCCP-Close packet; that receiving endpoint will expect to hold Time-Wait state after later receiving a DCCP-Reset.

```
  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 or 16 bytes)             /
 /           with Type=5 or 6 (DCCP-CloseReq or Close)           /
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |   Reserved    |            Acknowledgement Number             |
(+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+)Iff
(|       Acknowledgement Number (low bits)       |   Reserved    |)X=1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                     Options                   /   [padding]   |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

## 5.9. DCCP-Reset Packet Format

DCCP-Reset packets unconditionally shut down a connection. Every normal connection 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. The endpoint that receives a valid DCCP-Reset packet will hold Time-Wait state for the connection. The optional DCCP-Reset payload, if present, is a human-readable text string, preferably in English and encoded in Unicode UTF-8, that describes the error in more detail. DCCP-Reset packets MUST NOT be generated in response to received DCCP-Reset packets.

```
  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 or 16 bytes)             /
 /                   with Type=7 (DCCP-Reset)                    /
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |   Reserved    |             Acknowledgement Number            |
(+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+)Iff
(|      Acknowledgement Number (low bits)        |   Reserved    |)X=1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |    Reason     |    Data 1     |    Data 2     |    Data 3     |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                       Options                 /   [padding]   |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                          error text                           |
 |                             ...                               |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

**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 | 3.2 |
| 2 | Invalid Packet | packet type | N/A | N/A | 5.4 |
| 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 | 5.6 |
| 6 | Bad Service Code | N/A | N/A | N/A | 5.5 |
| 7 | Too Busy | N/A | N/A | N/A | 5.6 |
| 8 | Bad Init Cookie | N/A | N/A | N/A | 6.6 |
| 10 | Unanswered Challenge | N/A | N/A | N/A | 6.5.4 |
| 11 | Fruitless Negotiation | feature number | feature data (optional) | | 6.4.8 |
| 12 | Aggression Penalty | N/A | N/A | N/A | 9.2 |
| 13 | No Connection | N/A | N/A | N/A | 5.4 |
| 14 | Aborted | N/A | N/A | N/A | 5.4 |
| 15 | Extended Seqnos | N/A | N/A | N/A | 5.3 |
| 16 | Mandatory Failure | option number | option data (if any) | | 6.3 |
| 17-127 | Reserved | | | | |
| 128-255 | CCID-specific reasons | ... variable ... | | | 7.4 |

A DCCP-Reset packet completes every DCCP connection, whether the termination is clean (due to application close; Reset Reason "Closed") or unclean. Unlike TCP, which has two distinct termination mechanisms (FIN and RST), DCCP ends all connections in a uniform manner. This is justified because some responses to connection termination close are the same no matter whether termination was clean. For instance, the endpoint that receives a valid DCCP-Reset should hold Time-Wait state for the connection. Processors that must distinguish between clean and unclean termination can examine the Reset Reason.

DCCP implementations MUST transition to the CLOSED state after sending a DCCP-Reset packet.

## 5.10. 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 new address and port come from the packet's network header and generic DCCP header; the old address and port are defined through a Mobility ID, which must have been set earlier via a Mobility ID feature. The Mobility ID 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 or 16 bytes)             /
 /                    with Type=8 (DCCP-Move)                    /
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |   Reserved    |            Acknowledgement Number             |
(+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+)Iff
(|       Acknowledgement Number (low bits)       |   Reserved    |)X=1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                    Mobility ID (high bits)                    |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                    Mobility ID (low bits)                     |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |        Options, including Identification      /   [padding]   |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                             data                              |
 |                              ...                              |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

**Mobility ID: 64 bits**

The value of the sender's Mobility ID feature. This value uniquely identifies the current connection among the set of connections terminating at the receiver; it MUST have been set by the receiver in an earlier exchange.

**Options**

Every DCCP-Move packet MUST include a valid Identification option (see Section 6.5).

DCCP B MUST ignore the DCCP-Move if it has no record for the packet's Mobility ID; if the Identification option is not present or invalid; if the Sequence Number is not greater than GSR; or if the Acknowledgement Number is greater than GSS. DCCP B SHOULD NOT respond to invalid Moves with DCCP-Reset or DCCP-Ack packets, since any such response would leak information about the connection, such as the current sequence number, to a possibly malicious host. After receiving an invalid DCCP-Move, DCCP B MAY ignore subsequent DCCP-Move packets, valid or not, for a short period of time, such as one second or one round-trip time. This protects DCCP B against denial-of-service attacks from floods of invalid DCCP-Moves.

DCCP-Move packets do not follow the usual sequence-validity rules. This is to support endpoints that react to long bursts of loss by moving. Such moves will often happen after the endpoints get out of sync, causing DCCP-Move packets to frequently have inappropriate Sequence Numbers. But the usual DCCP-Sync mechanism is inappropriate in response to Moves, since it could leak sequence numbers to possibly malicious hosts. DCCP B MUST set its GSR variable to the Sequence Number on a valid DCCP-Move.

DCCP B SHOULD acknowledge valid DCCP-Move packets with DCCP-Ack or DCCP-DataAck packets. If DCCP B accepts the move, it MUST send this acknowledgement to the packet's network source address and DCCP Source Port; if it rejects the move, which it MAY do for any reason, it MUST send this acknowledgement to the old address and old port. The moving endpoint, DCCP A, can determine whether or not its move was accepted by checking the acknowledgement's destination address and Port.

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 source address and Source Port correspond to a valid connection, for which the Sequence Number and Acknowledgement Number fields are appropriate; the Identification option is valid for that connection; and the Mobility ID refers to that connection. It SHOULD respond by sending another acknowledgement, as allowed by the congestion control mechanism in use.

Once DCCP B receives a non-Move packet from DCCP A, it MUST choose a new Mobility ID for the connection and send a new Change R(Mobility ID) option to DCCP A. This reduces the risk of replay.

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.

### 5.11. DCCP-Sync Packet Format

DCCP-Sync packets are sent when the sequence numbers of the endpoints of a connection appear to have gotten out of sync. On receiving a valid DCCP-Sync packet, DCCP will update its GSR variable, thus restoring synchronization, and possibly send another DCCP-Sync packet to acknowledge the synchronization. DCCP-Sync 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 or 16 bytes)             /
 /                    with Type=9 (DCCP-Sync)                    /
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |   Reserved    |            Acknowledgement Number             |
(+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+)Iff
(|       Acknowledgement Number (low bits)       |   Reserved    |)X=1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                      Options                  /   [padding]   |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

## 6. Options and Features

All DCCP packets may contain options, which occupy space at the end of the DCCP header. Each option is a multiple of 8 bits in length. The combination of all options MUST add up to a multiple of 32 bits. Individual options are not padded to multiples of 32 bits, however; any option may begin on any byte boundary. All options are always included in the checksum.

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.

Options are processed sequentially, starting at the earliest option in the packet header.

The following options are currently defined:

| Type | Option Length | Meaning | Section Reference |
|---|---|---|---|
| 0 | 1 | Padding | 6.1 |
| 1 | 1 | Mandatory | 6.3 |
| 2 | 1 | Slow Receiver | 8.6 |
| 32 | variable | Ignored | 6.2 |
| 33 | variable | Change L | 6.4 |
| 34 | variable | Confirm L | 6.4 |
| 35 | variable | Change R | 6.4 |
| 36 | variable | Confirm R | 6.4 |
| 37 | variable | Init Cookie | 6.6 |
| 38 | variable | Ack Vector [Nonce 0] | 8.5 |
| 39 | variable | Ack Vector [Nonce 1] | 8.5 |
| 40 | variable | Data Dropped | 8.7 |
| 41 | 6 | Timestamp | 6.7 |
| 42 | 6-10 | Timestamp Echo | 6.9 |
| 43 | variable | Identification | 6.5.3 |
| 44 | variable | Challenge | 6.5.4 |
| 45 | 4 | Payload Checksum | 8.8 |
| 46 | 4-6 | Elapsed Time | 6.8 |
| 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 one DCCP converses with another, extended DCCP. Each Ignored option has one or more bytes of data. The first byte contains the offending option type; the second and subsequent, if present, contain the first bytes of the offending option's data. If the offending option had data, the Ignored option MUST include at least one byte of that data, but the Ignored option MUST NOT carry more Opt Data than the offending option had data.

Ignored options should preferably concern options sent on the packet acknowledged by the Acknowledgement Number. Packets without Acknowledgement Numbers (that is, DCCP-Request and DCCP-Data) SHOULD NOT carry Ignored options.

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

+--------+--------+--------+--------+--------
|00100000| Length |Opt Type|  Opt Data ...
+--------+--------+--------+--------+--------
 Type=32
```

### 6.3. Mandatory Option

The Mandatory option, with type 1, is a single byte option that indicates that the immediately following option is mandatory. If the receiving DCCP does not understand that following option, it MUST reset the connection with Reset Reason set to "Mandatory Failure". For instance, say DCCP A receives a packet with two options: a Mandatory option, and immediately following, another option O. Then DCCP A would reset the connection (rather than, for example, sending an Ignored(O) option) if it did not understand O's type; if it understood O's type, but not O's data; if O's data was invalid for O's type; if O was a feature negotiation option, and DCCP A did not understand the enclosed feature number; if DCCP A understood O, but chose not to perform the action O implies; and so forth.

```
+--------+
|00000001|
+--------+
  Type=1
```

### 6.4. 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.

Features are identified by feature number and owning endpoint. The notation (F,E) represents the feature with feature number F that is owned by DCCP E. A connection generally has two features for each feature number, one per endpoint (or, equivalently, one per half-connection). Given a feature owned by DCCP A, we call DCCP A the feature location and DCCP B the feature remote. Both endpoints keep track of the values of all features, since the point of feature negotiation is to ensure agreement.

Four options, Change L, Confirm L, Change R, and Confirm R, implement feature negotiation. The "L" options are sent by the feature location, the "R" options are sent by the feature remote. Change options initiate a negotiation, Confirm options complete the negotiation. Change options are retransmitted to ensure reliability.

Feature values MUST NOT change apart from feature negotiation. This property, retransmissions, and value priority rules ensure that both endpoints eventually agree on every feature's value.

Negotiations for multiple features may take place simultaneously. For instance, a packet may contain multiple Change options that refer to different features. The endpoints may also simultaneously open negotiations for the same feature; they will still agree on a single value.

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.

Here are three example feature negotiations for features located at DCCP B, the first two for the Congestion Control ID feature, the last for the Ack Ratio:

```
          DCCP A                          DCCP B
1. Change R(CCID, 2 3 1) --->
   ("2 3 1" is DCCP A's value preference list)
2.                      <--- Confirm L(CCID, 3, 3 2 1)
                        (3 is the negotiated value;
                        "3 2 1" is B's pref list)
          * agreement that (CCID,B) = 3 *

1.                  XXX <--- Change L(CCID, 3 2 1)
2.                           Retransmission:
                        <--- Change L(CCID, 3 2 1)
3. Confirm R(CCID, 3, 2 3 1) --->
          * agreement that (CCID,B) = 3 *

1. Change R(Ack Ratio, 3) --->
2.                      <--- Confirm L(Ack Ratio, 3)
       * agreement that (Ack Ratio,B) = 3 *
```

### 6.4.1. Value Types

The feature negotiation options are the same for every feature number, but the format for feature values, and the value priority rules that determine the result of a negotiation, differ from feature to feature. All current DCCP features fit one of two value types, non-negotiable ("NN") or server-priority ("SP"), although other value types are possible.

- **Non-negotiable features:** The feature value is a byte string. Each option contains exactly one feature value. The feature remote changes the value by sending Change R options. The feature location has no preferred value for the feature, and MUST accept the proposed value (as long as it is valid), responding with a Confirm L option containing the new value. Change L and Confirm R options MUST NOT be sent for non-negotiable features.
- **Server-priority features:** The feature value is a fixed-length byte string (length determined by the feature number). Each Change option contains a prioritized list of values, with the most preferred value coming first. Each Confirm option contains the confirmed value, followed by the confirmer's value preference list. The value priority rule is server priority: Given both preference lists, select the first entry in the server's list that also occurs in the client's list. If there is no shared entry, the connection MUST be reset with Reason set to Fruitless Negotiation. All four option types are meaningful for server-priority features.

  DCCP endpoints need not calculate their value preference lists before feature negotiation begins. Thus, a server might adjust its preference list based on the client's preference list, assuming the client opened the negotiation. Once a negotiation for a feature has begun, however, that feature's preference lists MUST remain stable until the negotiation has closed.

### 6.4.2. Feature Numbers

The first data byte of every Change 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 | Value Type | Initial Value | Section Reference |
|---|---|---|---|---|
| 1 | Congestion Control ID (CCID) | SP | 2 | 7 |
| 2 | ECN Capable | SP | 1 | 9.1 |
| 3 | Ack Ratio | NN | 2 | 8.3 |
| 4 | Use Ack Vector | SP | 0 | 8.4 |
| 5 | Mobility Capable | SP | 0 | 10.1 |
| 6 | Loss Window | NN | 1000 | 6.10 |
| 7 | Connection Nonce | NN | random | 6.5.2 |
| 8 | Identification Regime | SP | 1 | 6.5.1 |
| 9 | Mobility ID | NN | 0 | 10.2 |
| 128-255 | CCID-specific features | ? | ? | 7.4 |

### 6.4.3. Change L Option

DCCP A sends a Change L option to DCCP B to initiate a negotiation for a feature located at DCCP A. DCCP B SHOULD respond to a Change option for a known feature with a Confirm R option. In special circumstances, such as a Change option whose value is inappropriate for the listed feature number, 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 L option until it receives one of those responses. It could send at least one option per round-trip time, for instance, or it could add the Change L option to every Kth packet. DCCP A MAY reset the connection with Reason set to "Fruitless Negotiation" or "Feature Error" if retransmission fails (no meaningful response is received after 10 attempts or more). The format of the option's data ("Value or Values") depends on the feature's value type. Change L options are invalid for non-negotiable features.

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

An example Change L option follows.

**33,5,1,2,3**

I want to change my CC feature (feature number 1, a server-priority feature); my preferred values are 2 and 3, in that preference order.

### 6.4.4. Confirm L Option

DCCP A sends a Confirm L option to DCCP B in response to a valid Change R option sent by DCCP B. The Confirm L option will complete the negotiation for a feature located at DCCP A. Confirm L need not be retransmitted, since Change R will be retransmitted as necessary. Again, the format of "Value or Values" depends on the feature's value type.

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

Example Confirm L options follow.

**34,6,1,2,2,3**
I have changed my CC feature (feature number 1, a server-priority feature) to value 2; my preferred values are 2 and 3, in that preference order.

**34,9,7,239,48,2,188**
I have changed my Connection Nonce feature (feature number 7, a non-negotiable feature) to the 4-byte string 239,48,2,188.

### 6.4.5. Change R Option

DCCP A sends a Change R option to DCCP B to initiate a negotiation for a feature located at DCCP B. The possible responses to Change R are analogous to those for Change L (Confirm L, Ignored, or Reset). As with Change L, DCCP A SHOULD retransmit the Change R option until it receives a response, or the retransmission times out. Again, the format of "Value or Values" depends on the feature's value type.

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

Example Change R options follow.

**35,5,1,3,2**
Please change your CC feature (feature number 1, a server-priority feature); my preferred values are 3 and 2, in that preference order.

**35,9,7,239,48,2,188**
Change your Connection Nonce feature (feature number 1, a non-negotiable feature) to the 4-byte string 239,48,2,188.

### 6.4.6. Confirm R Option

DCCP A sends a Confirm R option to DCCP B in response to a valid Change L option sent by DCCP B. The Confirm R option will complete the negotiation for a feature located at DCCP B. Confirm R need not be retransmitted, since Change L will be retransmitted as necessary. Again, the format of "Value or Values" depends on the feature's value type.

```
+--------+--------+--------+--------+--------+--------
|00100100| Length |Feature#| Value or Values ...
+--------+--------+--------+--------+--------+--------
 Type=36
```

An example Confirm R option follows.

**36,6,1,2,3,2**
Change your CC feature (feature number 1, a server-priority feature) to 2; my preferred values are 3 and 2, in that preference order.

### 6.4.7. Unknown Features

If a DCCP receives a Change option referring to a feature number it does not understand, it SHOULD 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.)

### 6.4.8. State Diagram

These state diagrams present the legal transitions in a DCCP feature negotiation. They define a 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 the feature remote, DCCP B.

Each endpoint can be in one of three states, STABLE, CHANGING, and FAILED. The STABLE state means that a value is known for the feature and no negotiation is in progress. Every feature starts out in the STABLE state. The CHANGING state means that a negotiation started by this endpoint is in progress for the feature. This is the only state in which retransmissions happen. Finally, the FAILED state means that the other endpoint does not understand the feature in question.

Transitions between states are triggered by receiving a valid packet containing some valid negotiation option, or by an application or protocol event. Receiving a Change option causes the new feature value to be calculated, and a Confirm option sent. The details of this calculation, and the contents of Confirm, depend on the value type of the feature in question. Endpoints that receive valid Confirm options can simply trust the values they contain, or they could redo the feature value calculation; again, this is feature-specific.

```
            FEATURE LOCATION STATE DIAGRAM (DCCP A)

rcv Confirm R   app/protocol evt : snd Change L
: ignore        +----------------------------+
       +----+   |                            |
       |    v   |   rcv Confirm R            v
    +------------+  : accept value     +------------+
    |            |<--------------------|            |
    |   STABLE   |                     |  CHANGING  |------+
    |            |<--------------------|            |      |
    +------------+  rcv Change R       +------------+      |
        |     ^     : calc new value,      |     ^         |
        +-----+       snd Confirm L        +-----+         |
     rcv Change R                    timeout/rcv non-ack   |
     : calc new value,               : snd Change L        |
       snd Confirm L                                       |
                                rcv Ignored/timeout fails  |
                                : snd Reset/ignore/other   v
                                                      +----------+
                                                      |  FAILED  |
                                                      +----------+
```

```
             FEATURE REMOTE STATE DIAGRAM (DCCP B)

 rcv Confirm L   app/protocol evt : snd Change R
 : ignore        +---------------------------+
      +----+     |                           |
      |    v     |   rcv Confirm L           v
    +------------+  : calc new value  +------------+
    |            |<-------------------|            |
    |   STABLE   |                    |  CHANGING  |------+
    |            |<-------------------|            |      |
    +------------+  rcv Change L      +------------+      |
         |     ^    : calc new value,     |     ^         |
         +-----+      snd Confirm R       +-----+         |
    rcv Change L                     timeout/rcv non-ack  |
    : calc new value,                : snd Change R       |
      snd Confirm R                                       |
                              rcv Ignored/timeout fails   |
                              : snd Reset/ignore/other    v
                                                    +----------+
                                                    |  FAILED  |
                                                    +----------+
```

DCCP implementations MUST sanity-check options' data as appropriate for the feature before acting according to the diagram. For example, Ack Ratio takes two-byte, non-zero integer values, so a "Confirm(Ack Ratio, 0)" option is never valid. Server-priority features can tolerate some unknown values in the priority list, as long as the selected value is understood. Invalid options SHOULD cause a transition to the FAILED state, with an appropriate accompanying action, such as sending a reset with Reason set to "Feature Error".

The "`snd`" actions request the sending of a negotiation option. They do not force DCCP to immediately generate a packet; rather, they say which feature option SHOULD be sent on the next packet generated. A DCCP MAY choose to generate a packet, such as a DCCP-Ack, in response to some "`snd`" action, rather than piggyback on another packet. In some cases, this may be required---if adding an option would bump a packet over the PMTU, for instance. However, it MUST NOT generate a packet if doing so would violate the congestion control mechanism in use.

Retransmissions of Change options happen according to an exponential-backoff timer, and/or when the CHANGING DCCP realizes that the packet containing a Change option was not received. A Change option MAY additionally be piggybacked on other packets sent during the negotiation. After too many timer backoff events, or when an explicit Ignored option is received, the CHANGING DCCP MUST transition to the FAILED state, as shown. The CHANGING DCCP MUST NOT transition to the FAILED state simply because the other DCCP seems to be ignoring its Change options (for example, by acknowledging the packet containing the options, but not including a Confirm); reordering can cause this behavior even if the endpoint understands the options. The timeout value might initially be set to a small multiple of round-trip times (or 0.2 seconds, if no RTT is available). Backoff should be pinned at roughly 32 RTTs; timer failure should occur after at least 12 retransmissions.

Feature negotiation options for a given feature MUST be processed in increasing order by Sequence Number. Say that the last processed negotiation option for a feature (F,X) came on a packet with sequence number S. Then any negotiation options on received packets with Sequence Number less than or equal to S MUST be ignored. This requirement MAY be implemented per-feature, or implementations MAY compare against a single Sequence Number---the most recent negotiation option processed for any feature. Feature negotiation options on safely reordered packets (with last-negotiation-seqno < S < GSR) SHOULD be accepted, to provide some robustness against reordering.

Simultaneous negotiation problems can arise if value preferences change too frequently, particularly for server-priority features. A DCCP endpoint MUST NOT change its value preferences while in the CHANGING state: it MUST instead complete any extant negotiation, then open a new one.

If the result of some feature negotiation is that a feature has an unacceptable value---for example, for a server-priority feature, none of the client's choices were acceptable to the server, and the prior value is unacceptable to the client---a DCCP endpoint MAY reset the connection, with DCCP-Reset Reason set to "Fruitless Negotiation".

The CHANGING state signals that the relevant feature's value is in flux. DCCP MAY change its behavior when certain features are CHANGING---for example, by refusing to send data until reentering STABLE.

#### 6.4.9. Streamlined Negotiation

This section provides guidance for implementations that do not wish to implement full feature negotiation, although general-purpose DCCP implementations SHOULD implement negotiation fully.

Minimal DCCP implementations, such as those for embedded devices, might force all negotiation to take place on the first packet exchange. The DCCP-Request would contain Change R options for all server-located features, and Change L options for all client-located features; the DCCP-Response would Confirm each of these requests, or reset the connection if any Change was unexpected or unacceptable. Changes for CCID-specific features MUST follow Changes for the Congestion Control ID feature in the option list, since options are processed in order. Once the connection is set up, minimal implementations might respond to all feature negotiation options with Ignored, except that even minimal implementations SHOULD support "Change R(Ack Ratio)" and "Confirm L(Ack Ratio)".

Even general-purpose implementations might refuse to renegotiate the Congestion Control ID feature in the middle of the connection, by responding to "Change(CCID)" options with Ignored.

### 6.5. 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, either exchanged at the beginning of the connection or implicitly agreed upon.

### 6.5.1. Identification Regime Feature

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

The value of ID Regime is a two-byte number, so valid Confirm and Change(ID Regime) options take at least five bytes. Change options MAY list multiple ID Regimes in descending order of preference. This document defines two ID Regimes:

```
ID Regime   Meaning
---------   -------
    0       Null Regime
    1       MD5 Regime (default)
```

In the Null Regime, every Identification or Challenge option is invalid. The Null Regime makes it impossible for endpoints to get back into sync after bursts of loss larger than two-thirds of the Loss Window (Section In the MD5 Regime, which is the default, valid Identification and Challenge options contain an MD5 hash of the Connection Nonce feature values with some packet data. Applications preferring different security guarantees, particularly around mobility issues, may prefer to implement another identification algorithm and allocate a new ID Regime value for it.

If the endpoints cannot agree on mutually acceptable ID Regimes, the connection SHOULD be reset due to "Fruitless Negotiation".

### 6.5.2. 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, a value used by Identification Regime 1. Each endpoint SHOULD keep track of its own nonce and, via the Connection Nonce feature, the other endpoint's nonce. Connection Nonce is a non-negotiable feature.

The Connection Nonce feature takes arbitrary values of at least 4 bytes long. A Change or Confirm(Connection Nonce) option therefore takes at least 7 bytes.

Connection Nonce defaults to a random 8-byte string. To prevent spoofing, this string MUST NOT have any trivially predictable value. For example, it MUST NOT be set deterministically to zero, and it SHOULD change on every connection. DCCP endpoints MAY, however, exchange Connection Nonces via some mechanism other than the plaintext, snoopable Connection Nonce option. For example, two DCCPs might exchange nonces over a secure channel; or, assuming neither endpoint is behind a network address translator, they might encrypt the source and destination ports with a shared secret key.

### 6.5.3. 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:

```
+--------+--------+--------+--------+--------+--------
|00101011| Length |  Identification Data ...
+--------+--------+--------+--------+--------+--------
 Type=43
```

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 1, the default MD5 Regime.

The Identification data provided for the MD5 Regime consists of a 16-byte MD5 digest of: the 32-bit words in the DCCP header that include the Sequence and Acknowledgement Numbers (this will be words 3-4 or 3-6, depending on whether sequence numbers are extended); 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, and the option may only be provided on packets that contain Acknowledgement Numbers, such as DCCP-Ack. 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 and Acknowledgement Numbers 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 before transmitting a packet 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 + 8, 8);
      /* assuming 24-bit sequence numbers */
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);
```

### 6.5.4. 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 sent with a Challenge option depends on the active Identification Regime. For the default MD5 Regime (Regime 1), 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".

## 6.6. Init Cookie Option

This option is permitted in DCCP-Response, DCCP-Data, DCCP-Ack, and DCCP-DataAck messages. The server MAY include an Init Cookie option in its DCCP-Response. If so, then the client MUST echo the same Init Cookie option in each succeeding DCCP packet until one of those packets is acknowledged or the connection is reset. The server SHOULD design its Init Cookie format so that Init Cookies can be checked for tampering; it SHOULD respond to an tampered 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 code, 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.

Init Cookies are limited to at most 253 bytes in length.

```
+--------+--------+--------+--------+--------+--------
|00100101| Length |         Init Cookie Value   ...
+--------+--------+--------+--------+--------+--------
 Type=37
```

## 6.7. Timestamp Option

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

```
+--------+--------+--------+--------+--------+--------+
|00101001|00000110|          Timestamp Value          |
+--------+--------+--------+--------+--------+--------+
 Type=41  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.

## 6.8. Elapsed Time Option

This option is permitted in any DCCP packet that contains an Acknowledgement Number. It indicates how much time, in tenths of milliseconds, has elapsed since the packet being acknowledged---the packet with the given Acknowledgement Number---was received. The option may take 4 or 6 bytes, depending on the size of the Elapsed Time value. Elapsed Time helps correct round-trip time estimates when the gap between receiving a packet and acknowledging that packet may be long---in CCID 3, for example, where acknowledgements are sent infrequently.

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

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

The option data, Elapsed Time, represents an estimated upper bound on the amount of time elapsed since the packet being acknowledged was received, with units of tenths of milliseconds. If Elapsed Time is less than a second, the first, smaller 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. DCCP endpoints MUST NOT report Elapsed Times that are significantly larger than the true elapsed times. A connection MAY be reset, with Reason set to "Aggression Penalty", if one endpoint determines that the other is reporting a much-too-large Elapsed Time.

Elapsed Time is measured in tenths of milliseconds as a compromise between two conflicting goals. First, it provides enough granularity to reduce rounding error when measuring elapsed time over fast LANs. Second, Elapsed Time allows most reasonable elapsed times to fit into two bytes of data.

## 6.9. 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. Generally, a DCCP endpoint should send one Timestamp Echo option for each Timestamp option it receives; and it should send that option as soon as is convenient. The length of the option is between 6 and 10 bytes, depending on whether Elapsed Time is included and how large it is.

```
+--------+--------+--------+--------+--------+--------+
|00101010|00000110|           Timestamp Echo          |
+--------+--------+--------+--------+--------+--------+
 Type=42  Len=6

+--------+--------+------- ... -------+--------+--------+
|00101010|00001000|  Timestamp Echo   |  Elapsed Time   |
+--------+--------+------- ... -------+--------+--------+
 Type=42  Len=8       (4 bytes)

+--------+--------+------- ... -------+------- ... -------+
|00101010|00001010|  Timestamp Echo   |   Elapsed Time    |
+--------+--------+------- ... -------+------- ... -------+
 Type=42  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.

### 6.10. 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 R(Loss Window, W)" option to DCCP B to set DCCP B's Loss Window to W. Loss Window is non-negotiable.

The Loss Window feature takes 3- or 6-byte integer values, like DCCP sequence numbers. Change and Confirm options for Loss Window are therefore either 6 or 9 bytes long.

Loss Window defaults to 1000 for new connections. The Loss Window value is the total width of the loss window. The receiver positions the loss window asymmetrically around GSR, the greatest sequence number received, with one-third of the loss window width (rounded down) reserved for GSR and older sequence numbers and two-thirds reserved for newer sequence numbers. See Section 5.2.

## 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 ID features. Congestion Control ID 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 a "Change R(CCID, K)" option to DCCP A to ask A to use CCID K for its data packets. CCID is a server-priority feature.

The data byte of Congestion Control ID feature negotiation options form a list of acceptable CCIDs, sorted in descending order of priority. For example, the option "Change R(CCID, 1 2 3)" asks the receiver to use CCID 1 for its packets, although CCIDs 2 and 3 are also acceptable. (This corresponds to the bytes "35, 6, 1, 1, 2, 3": Change R option (35), option length (6), feature ID (1), CCIDs (1, 2, 3).) Similarly, "Confirm L(CCID, 1, 1 2 3)" tells the receiver that the sender is using CCID 1 for its packets, 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 a DCCP endpoint, that endpoint MUST send "Change(CCID)" options on its first packets, and MUST Reset the connection if the results of those negotiations are unacceptable.

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".

Many DCCP APIs will allow applications to suggest preferred CCIDs for sending and receiving data. Applications might be able to allow or prevent the use of CCID 1 for sending and receiving. For sending, however, it makes sense to let the code implementing a particular CCID silently suggest CCID 1 when appropriate.

CCID 1 places no restrictions on how often the HC-Receiver may send DCCP-Ack packets. This applies wherever we say "send a DCCP-Ack as allowed by the congestion control mechanism in use". A careful implementation SHOULD implement a liberal rate limit on DCCP-Acks to prevent ack storms, however.

## 7.2. TCP-like Congestion Control

CCID 2, TCP-like Congestion Control, 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 achieves maximum bandwidth over the long term, consistent with the use of end-to-end congestion control, but halves its congestion window in response to each congestion event. This leads to the abrupt rate changes typical of TCP. Applications should use CCID 2 if they prefer maximum bandwidth utilization to steadiness of rate. This is often the case for applications that are not playing their data directly to the user. For example, a hypothetical application that transferred files over DCCP, using application-level retransmissions for lost packets, would prefer CCID 2 to CCID 3. On-line games may also prefer CCID 2.

CCID 2 is further described in [CCID 2 PROFILE].

## 7.3. TFRC Congestion Control

CCID 3 denotes TCP-Friendly Rate Control (TFRC), an equation-based rate-controlled congestion control mechanism. TFRC is designed to be reasonably fair when competing for bandwidth with TCP-like flows, where a flow is "reasonably fair" if its sending rate is generally within a factor of two of the sending rate of a TCP flow under the same conditions. However, TFRC has a much lower variation of throughput over time compared with TCP, which makes CCID 3 more suitable than CCID 2 for applications such as telephony or streaming media where a relatively smooth sending rate is of importance.

CCID 3 is further described in [CCID 3 PROFILE]. The TFRC congestion control algorithms were initially described in [RFC 3448].

## 7.4. CCID-Specific Options, Features, and Reset Reasons

Option types, feature numbers, and Reset Reasons 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 and features introduced by later versions of this specification.

Any packet may contain information meant for either half-connection, so CCID-specific option types, feature numbers, and Reset Reasons 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.
- Reset Reasons 128 through 191 indicate that the HC-Sender reset the connection (most likely because of some problem with acknowledgements sent by the HC-Receiver); Reset Reasons 192 through 255 indicate that the HC-Receiver reset the connection (most likely because of some problem with data packets sent by the HC-Sender).
- Finally, feature numbers 128 through 191 are used for features located at the HC-Sender; feature numbers 192 through 255 are for features located at the HC-Receiver. Since Change L and Confirm L options for a feature are sent by the feature location, we know that any Change L(128) option was sent by the HC-Sender, while any Change L(192) option was sent by the HC-Receiver. Similarly, Change R(128) options are sent by the HC-Receiver, while Change R(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:

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

B > A   128                        B-to-A     5
B > A   192                        A-to-B     4
B > A   Change L(128, ...)         B-to-A     5
B > A   Change R(192, ...)         B-to-A     5
B > A   Confirm L(128, ...)        B-to-A     5
B > A   Confirm R(192, ...)        B-to-A     5
B > A   Change R(128, ...)         A-to-B     4
B > A   Change L(192, ...)         A-to-B     4
B > A   Confirm R(128, ...)        A-to-B     4
B > A   Confirm L(192, ...)        A-to-B     4
```

CCID-specific options and features have no clear meaning when a nontrivial negotiation for the relevant CCID is in progress. This can happen when a CCID-specific option follows a Change(CCID) option. Say the Change option prefers CCID X. Then the negotiation is nontrivial if and only if its result is not X. CCID-specific options and features MUST be ignored during a nontrivial CCID negotiation---for instance, by responding Ignored options---except that Mandatory CCID-specific options and features MUST induce a DCCP-Reset with Reason "Mandatory Error".

# 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). Detailed acknowledgement options are not necessarily required on every packet that carries an Acknowledgement Number, however.

## 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, and periodically retransmit, 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 the maximum of 0.2 seconds and 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 at which 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 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 rough 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 R(Ack Ratio)" option to DCCP B to change DCCP B's ack ratio. Ack Ratio is a non-negotiable feature.

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.

Implementations should treat Ack Ratio as a loose guideline. For instance, a DCCP endpoint might implement a delayed acknowledgement timer like TCP's, whereby each packet is acknowledged within at most T seconds of its receipt. (In TCP, T is commonly set to 200 milliseconds.) This is explicitly allowed even though it might lead to sending more acknowledgement packets than Ack Ratio would suggest. Particular algorithms for setting and using Ack Ratio are discussed in the relevant CCID drafts.

## 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 R(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 is a server-priority feature.

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:

```
+--------+--------+--------+--------+--------+--------
|0010011?| Length |SSLLLLLL|SSLLLLLL|SSLLLLLL|  ...
+--------+--------+--------+--------+--------+--------
Type=38/39         \___________ Vector ___________...
```

The two Ack Vector options (option types 38 and 39) 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.

Appendix A provides a non-normative description of the details of DCCP acknowledgement handling, in the context of an abstract Ack Vector implementation.

### 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 |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. For an old state of 0 (received non-marked) and received state of 1 (received ECN marked), the packet's new state may be set to either 0 or 1. The HC-Sender implementation will be indifferent to ack reordering if it chooses new state 1 for that cell.

The HC-Receiver should collect information about received packets, which it will eventually report to the HC-Sender on one or more acknowledgements, according to the following table:

```
            Received Packet
               0   1   3
             +---+---+---+
           0 | 0 |0/1| 0 |
   Stored    +---+---+---+
           1 |0/1| 1 | 1 |
    State    +---+---+---+
           3 | 0 | 1 | 3 |
             +---+---+---+
```

This table equals the sender's table, except that when the stored state is 1 and the received state is 0, the receiver is allowed to switch its stored state to 0.

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 Acknowledgement Window. Generally, every Ack Vector generated by the HC-Receiver will cover the whole Acknowledgement Window: Ack Vector acknowledgements are cumulative. (This simplifies Ack Vector maintenance at the HC-Receiver; see Section A, 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, with or without Data Dropped, 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. The sending application should not react to Slow Receiver by sending more data, however. For example, the optimal response to a CPU-bound receiver might be to increase the sending rate, by switching to a less-compressed sending format, since a highly-compressed data format might overwhelm a slow CPU more seriously than the higher memory requirements of a less-compressed data format. The Slow Receiver option is not appropriate for this case; a CPU-bound receiver should not ask for Slow Receiver options to be sent.

Slow Receiver implements a portion of TCP's receive window functionality. We believe receiver operating systems and applications will find it 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 respond to data-dropped packets less severely than to lost or marked packets. For instance, a windowed mechanism might subtract a constant value from its congestion window, rather than cut it in half.

Data Dropped lets a sender differentiate between different kinds of loss (network and endpoint), but it does not allow total freedom in how to react. The congestion control response to a Data Dropped packet must be approved by the IETF. Each congestion control mechanism MUST react to a Data Dropped packet as if the packet were ECN marked, unless explicitly specified otherwise.

If a received packet's payload is dropped for one of the reasons listed below, this SHOULD be reported using a Data Dropped option. Alternatively, the receiver MAY choose to report as "received" only those packets whose payloads were not dropped, subject to the constraint that packets not reported as received MUST NOT have had their options processed.

The option's data looks like this:

```
+--------+--------+--------+--------+--------+--------
|00101000| Length | Block  | Block  | Block  |  ...
+--------+--------+--------+--------+--------+--------
 Type=40          \___________ 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|DrpCd|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 Drop Code [DrpCd] 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 Codes 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 because the application is no longer listening.

**2** Packet data dropped in the receive buffer.

**3** Packet data dropped due to corruption.

**4-6** Reserved.

**7** Packet data corrupted, but delivered to the application anyway.

For example, if a Data Dropped option contains the decimal values 0,160,3,162, 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 Code 2, 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 Code 2, 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".

A DCCP application interface should let receiving applications specify the Drop Codes corresponding to received packets. For example, this would let applications calculate their own checksums, but still report "dropped due to corruption" packets via the Data Dropped option. The interface should not let applications reduce the "seriousness" of a packet's Drop Code; for example, the application should not be able to upgrade a packet from delivered corrupt (Drop Code 7) to delivered normally (no Drop Code).

### 8.7.1. Data Dropped and Normal Congestion Response

When deciding on a response to a particular acknowledgement or set of acknowledgements containing Data Dropped packets, a congestion control mechanism MUST consider dropped packets and ECN marks (including ECN-marked packets that are included in Data Dropped), as well as the Data Dropped packets. For window-based mechanisms, the valid response space is defined as follows.

Assume an old window of W. Independently calculate a new window W_new1 that assumes no packets were Data Dropped (so W_new1 contains only the normal congestion response), and a new window W_new2 that assumes no packets were lost or marked (so W_new2 contains only the Data Dropped response). We are assuming that Data Dropped recommended a reduction in congestion window, so W_new2 < W.

Then the actual new window W_new MUST NOT be larger than the minimum of W_new1 and W_new2; and the sender MAY combine the two responses, by setting W_new = W + min(W_new1 - W, 0) + min(W_new2 - W, 0).

Non-window-based congestion control mechanisms MUST behave analogously.

### 8.7.2. Particular Drop Codes

Drop Code 0 ("protocol constraints") does not indicate any kind of congestion, so the sender's CCID SHOULD react to non-marked packets with Drop Code 0 as if they were received. However, the sending DCCP SHOULD NOT send more data until it believes the relevant protocol constraint has passed.

Drop Code 1 ("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 Code 1 SHOULD be sent whenever received data is ignored due to a non-listening application. Once a DCCP reports Drop Code 1 for a packet, it SHOULD report Drop Code 1 for every succeeding data packet on that half-connection; once a DCCP receives a Drop State 1 report, it SHOULD expect that no more data will ever be delivered to the other endpoint's application, so it SHOULD NOT send more data. A DCCP receiving Drop Code 1 MAY report this event to the application. (Previous versions of this specification used a "Buffer Closed" option instead of Drop Code 1.)

Drop Code 2 ("receive buffer drop") indicates congestion inside the receiving host. Every packet newly acknowledged as Drop Code 2 SHOULD reduce the sender's instantaneous rate by one packet per round trip time, using whatever mechanism is appropriate for the relevant CCID. Further details may be available in CCID documents.

## 8.8. Payload Checksum Option

The Payload Checksum option holds the Internet checksum (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 nonzero Checksum Coverage, 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 Coverage 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 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 Code 3) 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 Code 7) 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.

A packet processor with access to link-layer error detection mechanisms might explicitly set Payload Checksum to zero when the link layer reported that a portion of the payload was corrupted. No actual Internet checksum has value zero, so this reliably informs the receiver that the payload is corrupt.

Note that Payload Checksum's value is included in the header checksum.

The Internet checksum used by the Payload Checksum option is generally considered weak, but it has the advantage that all IP processors can already calculate it. Applications desiring a stronger Payload Checksum should either send a checksum with the payload (reporting any checksum violations via the Data Dropped API), or propose a new checksum option.

See Section B.1 for a discussion of the issues related to the use of this option.

# 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 "Change L(ECN Capable, 0)" option to DCCP B to inform B that A cannot read ECN bits. The ECN Capable feature is a server-priority feature.

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 "Change L(ECN Capable, 0)" options to the other endpoint until acknowledged (by "Confirm R(ECN Capable, 0)") or the connection closes. Furthermore, it MUST NOT accept any data until the other endpoint sends "Confirm R(ECN Capable, 0)". It SHOULD send Data Dropped options on its acknowledgements, with Drop Code 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 38) and Ack Vector [Nonce 1] (option 39), 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 38) 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 R(Mobility Capable, 1)" option to DCCP A to inform it that B might like to move later. Mobility Capable is a server-priority feature.

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. Mobility ID

A DCCP uses the Mobility ID feature to inform its partner of a 64-bit number that will act as identification, should the partner need to change its address and/or port during the course of the connection.

Mobility ID has feature number 9. The Mobility ID feature located at DCCP A is the identifier that A will use on DCCP-Move packets it sends to B. DCCP B sends a "Change R(Mobility ID, N)" option to DCCP A to inform it that of the ID A has chosen for B's use. Mobility ID is a non-negotiable feature.

A Mobility ID feature contains eight bytes of data. The feature remote, say DCCP A, chooses the value of Mobility ID to uniquely identify a connection; its value must not equal the value of any other Mobility ID currently maintained by DCCP A. For security, DCCP A MUST choose Mobility ID randomly. Furthermore, it MUST reassign Mobility ID after each successful move by DCCP B, and it MAY reassign Mobility ID more frequently.

A new connection starts with Mobility ID 0 for both DCCPs. Zero is not a valid Mobility ID.

## 10.3. Security

The DCCP mobility mechanism, like DCCP in general, does not provide cryptographic security guarantees. Nevertheless, mobile hosts must use valid Mobility IDs 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 Mobility IDs 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.

## 10.4. 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".

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

## 10.5. 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 5.10 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 should not let loss events on packets from the old address/port pair affect the new congestion control state.

# 11. Maximum Packet Size

A DCCP implementation MUST maintain the maximum packet size (MPS) allowed for each active DCCP session. The MPS is influenced by the maximum packet size allowed by the current congestion control mechanism (CCMPS), the maximum packet size supported by the path's links (PMTU, the Path Maximum Transfer Unit) [RFC 1191], and the lengths of the IP and DCCP headers.

A DCCP application interface should let the application discover DCCP's current MPS. DCCP applications should use the API to discover the MPS. Generally, the DCCP implementation will refuse to send any packet bigger than the MPS, returning an appropriate error to the application.

A DCCP interface may allow applications to request that packets larger than PMTU be fragmented. This only matters when CCMPS > PMTU; packets larger than CCMPS MUST be rejected regardless. Fragmentation should not be the default. The rest of this section assumes the application has not requested fragmentation.

The MPS reported to the application SHOULD be influenced by the size expected to be required for DCCP headers and options. If the application provides data that, when combined with the options the DCCP implementation would like to include, would exceed the MPS, the implementation should either send the options on a separate packet (such as a DCCP-Ack) or lower the MPS, drop the data, and return an appropriate error to the application.

The PMTU SHOULD be initialized from the interface MTU that will be used to send packets. The MPS will be initialized with the minimum of the PMTU and the CCMPS, if any.

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.

As specified in [RFC 1191], when a router receives a packet with DF set that is larger than the next link's MTU, 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 MPS, 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 MPS. When this occurs, it is possible that the kernel has some packets buffered for transmission that are smaller than the old MPS, but larger than the new MPS. 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.

A DCCP implementation may allow the application to occasionally request that PMTU discovery be performed again. This will reset the PMTU to the outgoing interface's MTU. Such requests SHOULD be rate limited, to one per two seconds, for example. A successful DCCP-Move will also reset the PMTU.

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 link MTU it exceeded.

With application support, DCCP also allows for upward probing of PMTU [PMTUD]: the application would start by sending small packets, then gradually increase their sizes. A DCCP implementation supporting this upward probing MAY treat the loss of packets after a packet-size increase as an indication of MTU limitation, rather than congestion. XXX

## 12. Middlebox Considerations

This section describes properties of DCCP that firewalls, network address translators, and other middleboxes should consider, including parts of the packet that middleboxes should not change. The intent is to draw attention to aspects of DCCP that may be useful, or dangerous, for middleboxes, or that differ significantly from TCP.

The Service Code field in DCCP-Request packets provide information that may be useful for stateful middleboxes. With Service Code, a middlebox can tell what protocol a connection will use without relying on port numbers. Middleboxes can disallow attempted connections accessing unexpected services by sending a DCCP-Reset with Reason set to "Bad Service Code". Middleboxes probably shouldn't modify the Service Code, unless they are really changing the service a connection is accessing.

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.

Modifying DCCP Sequence Numbers and Acknowledgement Numbers is more tedious and dangerous than modifying TCP sequence numbers. A middlebox that added packets to, or removed packets from, a DCCP connection would have to modify, at least: (1) acknowledgement options, such as Ack Vector; (2) CCID-specific options, such as TFRC's Loss Intervals; and (3) Identification options---for example, the default MD5 Identification Regime includes sequence numbers in its cryptographic hash. On ECN-capable connections, the middlebox would have to keep track of ECN Nonce information for packets it introduced or removed, so that the relevant acknowledgement options continued to have correct ECN Nonce Echoes, or risk the connection being reset for "Aggression Penalty". We therefore recommend that middleboxes not modify packet streams by adding or removing packets.

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 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. This might involve fragmenting the packet.)

Middleboxes may, of course, reset connections in progress. Clearly this requires inserting a packet into one or both packet streams, but the difficult issues do not arise.

DCCP is somewhat unfriendly to "connection splicing" [SHHP00], in which clients' connection attempts are intercepted, but possibly later "spliced in" to external server connections via sequence number manipulations. A connection splicer at minimum would have to ensure that the spliced connections agreed on all relevant feature values, which might take some renegotiation.

Middleboxes that want to trivially support the MD5 Identification Regime (Section 6.5.3) should not alter packets' Sequence Number, Type, # NDP, Acknowledgement Number, and Reserved 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.

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 3550], 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.

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 slightly more than 4 bytes per packet relative to RTP-over-UDP, and that eliminating the redundancy would not reduce the overhead.

First, consider acknowledgements. Both RTP and DCCP report feedback about loss rates to data senders, via Real-Time Control Protocol Sender and Receiver Reports (RTCP SR/RR packets) and via DCCP acknowledgement options. These feedback mechanisms are potentially redundant. However, RTCP SR/RR packets contain information not present in DCCP acknowledgements, such as "interarrival jitter", and DCCP's acknowledgements contain information not transmitted by RTCP, such as the ECN Nonce Echo. Neither feedback mechanism encompasses the other.

Sending both types of feedback isn't particularly costly either. RTCP reports are sent relatively infrequently: once every 5 seconds, for low-bandwidth flows. In DCCP, some feedback mechanisms are expensive---Ack Vector, for example, is frequent and verbose---but others are relatively cheap: CCID 3 (TFRC) acknowledgements take between 16 and 32 bytes of options sent once per round trip time. (Reporting less frequently than once per RTT would make congestion control less responsive to loss.) We therefore conclude that acknowledgement overhead in RTP-over-DCCP is not significantly higher than for RTP-over-UDP, at least for CCID 3.

One clear redundancy can be addressed at the application level. The verbose packet-by-packet loss reports sent in RTCP Extended Reports (RTCP XR) Loss RLE Blocks can be derived from DCCP's Ack Vector options. (The converse is not true, since Loss RLE Blocks contain no ECN information.) Since DCCP implementations should provide an API for application access to Ack Vector information, RTP-over-DCCP applications might request either DCCP Ack Vectors or RTCP Extended Report Loss RLE Blocks, but not both.

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 cannot be inferred from the DCCP sequence number either; for instance, RTP sequence numbers might be sent out of order. Furthermore, removing RTP's sequence number would not save any header space because of alignment issues. We therefore recommend that RTP transmitted over DCCP use the same headers currently defined. The 4 byte header cost is a reasonable tradeoff for DCCP's congestion control features and access to ECN. Truly bandwidth-starved endpoints should use header compression.

## 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 mobility-incapable 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) mechanism provide this guarantee. We also avoid leaking sequence numbers to possibly malicious endpoints. This is why invalid DCCP-Moves are ignored rather than reset, for example.

### 16.1. Security Considerations for Mobility

Mobility slightly changes this security guarantee by introducing a new mechanism by which an attacker can hijack a connection. This mechanism, DCCP-Move, has the unfortunate property that, given a successful attack, the victim could not realize that the connection has been stolen---its connection would simply be reset unexpectedly.

Nevertheless, a DCCP attacker still must snoop on data packets to get any reasonable probability of success. Specifically, an attacker can send a valid DCCP-Move packet if it can guess a valid Mobility ID AND it can generate valid Identification options. DCCP-Move packets need not contain valid Sequence or Acknowledgement Numbers, since a move might often follow a long burst of loss, so endpoints must choose these values well to prevent attack. Randomly choosing Connection Nonces and Mobility IDs should suffice, although we are concerned about the fact that Mobility IDs do not expire like sequence numbers do [[XXX]].

### 16.2. Security Considerations for Partial Checksums

The partial checksum facility has separate security impact, particularly in its interaction with authentication and encryption mechanisms. The impact is the same in DCCP as in the UDP-Lite protocol, and what follows was adapted from the corresponding text in the UDP-Lite specification [UDP-LITE].

When a DCCP packet's Checksum Coverage field is not zero, the uncovered portion of a packet may change in transit. This is contrary to the idea behind most authentication mechanisms: authentication succeeds if the packet has not changed in transit. Unless authentication mechanisms that operate only on the sensitive part of packets are developed and used, authentication will always fail for partially-checksummed DCCP packets whose uncovered part has been damaged.

The IPsec integrity check (Encapsulation Security Protocol, ESP, or Authentication Header, AH) is applied (at least) to the entire IP packet payload. Corruption of any bit within that area will then result in the IP receiver discarding a DCCP packet, even if the corruption happened in an uncovered part of the DCCP payload.

When IPsec is used with ESP payload encryption, a link can not determine the specific transport protocol of a packet being forwarded by inspecting the IP packet payload. In this case, the link MUST provide a standard integrity check covering the entire IP packet and payload. DCCP partial checksums provide no benefit in this case.

Encryption (e.g., at the transport or application levels) may be used. Note that omitting an integrity check can, under certain circumstances, compromise confidentiality [BEL98].

If a few bits of an encrypted packet are damaged, the decryption transform will typically spread errors so that the packet becomes too damaged to be of use. Many encryption transforms today exhibit this behavior. There exist encryption transforms, stream ciphers, which do not cause error propagation. Proper use of stream ciphers can be quite difficult, especially when authentication-checking is omitted [BB01]. In particular, an attacker can cause predictable changes to the ultimate plaintext, even without being able to decrypt the ciphertext.

## 17. IANA Considerations

DCCP introduces several sets of numbers whose values should be allocated by IANA. The following sets of numbers should require an IETF standards-track specification as a prerequisite for new registrations.

- DCCP Packet Types 9 through 15 (Section 5.1).
- 8-bit DCCP-Reset Reasons (Section 5.9).
- 8-bit DCCP Option Types (Section 6). The CCID-specific options 128 through 255 need not be allocated by IANA, although particular CCIDs may request that IANA allocate their CCID-specific options.
- 8-bit DCCP Feature Numbers (Section 6.4). The CCID-specific features 128 through 255 need not be allocated by IANA, although particular CCIDs may request that IANA

allocate their CCID-specific features.

- 8-bit DCCP Congestion Control Identifiers (CCIDs) (Section 7).
- 16-bit Identification Regimes, for use with DCCP Identification and Challenge options (Section 6.5).
- Ack Vector States (Section 8.5). Only State 2 remains unallocated.
- Data Dropped Drop Codes 4 through 6 (Section 8.7).

32-bit Service Codes (Section 5.5), which are not specific to DCCP, will require more liberal registration rules. Service Codes are meant to correspond to application-level services. For example, there might be a Service Code for HTTP connections, one for FTP control connections, and one for FTP data connections. However, a special-purpose Web server might use a Service Code different from HTTP's to indicate its function. We suggest that IANA allocate Service Codes to anyone who asks, subject to the following guidelines.

- No specification, standards-track or otherwise, is required to request a Service Code.
- Service Codes should be allocated one at a time, or in small blocks. A particular intended service should be described, in a short English phrase, before a Service Code can be allocated.
- IANA should maintain an association of Service Codes to the corresponding short English phrases.
- Users may request specific Service Code values. The requested values should be assigned first-come first-serve. We suggest that users request Service Codes that can be interpreted as meaningful four-byte ASCII strings. Thus, the "Frobodyne Plotz Protocol" might correspond to "fdpz", or the number 1717858426. The canonical interpretation of a Service Code field is numeric.
- The subset of Service Codes in which the high-order byte has a value between 65 and 90, inclusive---the capital letters in ASCII---should be reserved for international standard or standards-track specifications, IETF or otherwise.
- Furthermore, the subset of Service Codes in which the high-order byte has the value 63---ASCII '?'---should never be allocated. These Service Codes are reserved for private use.
- Service Code 0 should never be allocated either. It represents the absence of a meaningful Service Code.

This design for Service Code allocation is based on the allocation of 4-byte identifiers for Macintosh resources, PNG chunks, and TrueType and OpenType tables.

Finally, 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. 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 especially thank the DCCP expert reviewers: Greg Minshall, Eric Rescorla, and Magnus Westerlund for detailed written comments and problem spotting, and Rob Austein and Steve Bellovin for verbal comments and written notes.

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, Gorry Fairhurst, Derek Fawcus, David Timothy Fleeman, John Loughney, Ghyslain Pelletier, Tom Phelan, Stanislav Shalunov, Yufei Wang, and Michael Welzl. In particular, Michael Welzl suggested the Payload Checksum option.

# A. Appendix: Ack Vector Implementation Notes

This appendix 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 buffer grows from right to left. The implementation maintains five variables, aside from the buffer contents:

- "buf_head" and "buf_tail", which mark the live portion of the buffer.
- "buf_ackno", the Acknowledgement Number of the most recent packet acknowledged in the buffer. This corresponds to the "head" pointer.
- "buf_nonce", the one-bit sum (exclusive-or, or parity) of the ECN Nonces received on all packets acknowledged by the buffer with State 0.

We draw acknowledgement buffers like this:

```
+--------------------------------------------------------------------+
|S,L|S,L|S,L|S,L|   |   |   |   |S,L|S,L|S,L|S,L|S,L|S,L|S,L|S,L|
+--------------------------------------------------------------------+
                ^                            ^
             buf_tail       buf_head, buf_ackno = A     buf_nonce = 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, and will add an annotation showing the Acknowledgement Number for the last live byte in the buffer. For example:

```
  +-----------------------------------------------+
A |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    BN[E]
  +-----------------------------------------------+
```

Here, buf_nonce equals E and buf_ackno equals A. This smaller Example Buffer contains actual data.

```
   +--------------------------+
10 |0,0|3,0|3,0|3,0|0,4|1,0|0,0| 0    BN[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.

Additionally, the HC-Receiver must keep some information about the Ack Vectors it has recently sent. For each packet sent carrying an Ack Vector, it remembers four variables:

- "ack_seqno", the Sequence Number used for the packet. This is an HC-Receiver sequence number.
- "ack_ptr", the value of buf_head at the time of acknowledgement.
- "ack_ackno", the Acknowledgement Number "A" used for the packet. This is an HC-Sender sequence number. Since acknowledgements are cumulative, this single number completely specifies all necessary information about the packets acknowledged by this Ack Vector.
- "ack_nonce", the one-bit sum of the ECN Nonces for all State 0 packets in the buffer from Head to "A", inclusive. Initially, this equals the Nonce Echo of the acknowledgement's Ack Vector (or, if the ack packet contained more than one Ack Vector, the exclusive-or of all the acknowledgement's Ack Vectors), but it can change as information about old acknowledgements is removed, or as old packets arrive (so they change from State 3 or State 1 to State 0).

## A.1. Packet Arrival

This section describes how the HC-Receiver updates its acknowledgement buffer as packets arrive from the HC-Sender.

### A.1.1. New Packets

When a packet with Sequence Number greater than buf_ackno arrives, the HC-Receiver updates buf_head (by moving it to the left appropriately), buf_ackno (which is set to the new packet's Sequence Number), and possibly buf_nonce (if the packet arrived unmarked with ECN Nonce 1), in addition to the buffer itself. For example, if HC-Sender packet 11 arrived ECN marked, the Example Buffer above would enter this new state (changes are marked with stars):

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

Of course, the new packet's sequence number might not equal the expected sequence number. In this case, the HC-Receiver will 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 if 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     BN[0]
         ** +*******----------------------------+          *
```

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 (by increasing run lengths when possible), transfer its state to a larger buffer, or, as a last resort, drop all received packets, without processing them whatsoever, until its buffer shrinks again.

### A.1.2. Old Packets

When a packet with Sequence Number S arrives, and S <= buf_ackno, the HC-Receiver will scan the table for the byte corresponding to S. (Indexing structures could reduce the complexity of this scan.) If S was previously lost (State 3), and it was stored in a byte with run length 0, 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     BN[1]
                    +---------*------------------+
```

If S was not marked as lost, or if it was 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 is allowed then.) If S's buffer byte has a non-zero run length, then the buffer might need be reshuffled to make space for one or two new bytes.

The ack_nonce fields may also need manipulation when old packets arrive. In particular, when S transitions from State 3 or State 1 to State 0, and S had ECN Nonce 1, then the implementation should flip the value of ack_nonce for every acknowledgement with ack_ackno >= S.

It is impossible with this data structure to shift packets from State 0 to State 1, since the buffer doesn't store individual packets' ECN Nonces.

## A.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.

If the acknowledgement fits in one Ack Vector, that Ack Vector's Nonce Echo simply equals buf_nonce. For multiple Ack Vectors, more care is required. The Ack Vectors should be split at points corresponding to previous acknowledgements, since the stored ack_nonce fields provide enough information to calculate correct Nonce Echoes. The implementation should therefore acknowledge data at least once per 253 bytes of buffer state. (Otherwise, there'd be no way to calculate a Nonce Echo.)

For each acknowledgement it sends, the HC-Receiver will add an acknowledgement record. ack_seqno will equal the HC-Receiver sequence number it used for the ack packet; ack_ackno will equal buf_ackno; and ack_nonce will equal buf_nonce.

## A.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 finds the acknowledgement record R with the appropriate ack_seqno, then:

- Sets buf_tail to R.ack_ptr + 1.
- If R.ack_nonce is 1, it flips buf_nonce, and the value of every ack_nonce for later ack record.
- Throws away R and every preceding ack record.

(The HC-Receiver 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) ack_seqno = 59, ack_ackno = 3, ack_nonce = 1.

(2) ack_seqno = 60, ack_ackno = 10, ack_nonce = 0.

Say the HC-Receiver then received a DCCP-DataAck packet with Acknowledgement Number 59 from the HC-Sender. 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     BN[0]
       +------------------*+  *       *
```

The tail byte's run length was adjusted, since packet 3 was in the middle of that byte. Since R.ack_nonce was 1, the buf_nonce field was flipped, as were the ack_nonce fields for later acknowledgements (here, the HC-Receiver Ack 60 record, not shown, has its ack_nonce set to 1). The HC-Receiver can also throw away stored information about HC-Receiver Ack 59 and any earlier acknowledgements.

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      BN[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) ack_seqno = 59, ack_ackno = 3, ack_nonce = 1.

(2) ack_seqno = 60, ack_ackno = 3, ack_nonce = 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.

## A.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.

# B. Appendix: Design Motivation

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

## B.1. CsCov 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 MUST 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.

# 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.

# Informative References

[BB01] S.M. Bellovin and M. Blaze. Cryptographic Modes of Operation for the Internet. 2nd NIST Workshop on Modes of Operation, August 2001.

[BEL98] S.M. Bellovin. Cryptography and the Internet. Proc. CRYPTO '98 (LNCS 1462), pp46-55, August, 1988.

[CCID 2 PROFILE] S. Floyd and E. Kohler. Profile for DCCP Congestion Control ID 2: TCP-like Congestion Control. draft-ietf-dccp-ccid2-04.txt, work in progress, October 2003.

[CCID 3 PROFILE] S. Floyd, E. Kohler, and J. Padhye. Profile for DCCP Congestion Control ID 3: TFRC Congestion Control. draft-ietf-dccp-ccid3-04.txt, work in progress, October 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.

[PMTUD] Matt Mathis, John Heffner, and Kevin Lahey. Path MTU Discovery. draft-ietf-pmtud-method-00.txt, work in progress, October 2003.

[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.

[RFC 3448] M. Handley, S. Floyd, J. Padhye, and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 3448.

[RFC 3550] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550.

[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.

[SYNCOOKIES] Daniel J. Bernstein. SYN Cookies. http://cr.yp.to/syncookies.html, as of July 2003.

[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-02.txt, work in progress, August 2003.

## 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

## Full Copyright Statement