Internet Engineering Task Force
INTERNET-DRAFT
draft-ietf-dccp-spec-06.ps
Expires: August 2004

Eddie Kohler
UCLA
Mark Handley
UCL
Sally Floyd
ICIR

16 February 2004

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 unicast 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-05.txt:

* Organization overhaul.

* Add pseudocode for event processing.

* Remove # NDP; replace with Ack Count.

* Remove Identification, Challenge, ID Regime, and Connection Nonce.

* Data Checksum (formerly Payload Checksum) uses a 32-bit CRC.

* Switch location of non-negotiable features to clarify presentation; now the feature location controls its value.

* Rename "value type" to "reconciliation rule".

* Rename "Reset Reason" to "Reset Code".

* Mobility ID becomes 128 bits long.

* Add probabilities to Mobility ID discussion.

* Add SyncAck.

Table of Contents

1. Introduction

This document describes the Datagram Congestion Control Protocol (DCCP), a transport protocol that implements a congestion-controlled, bidirectional stream of unreliable datagrams. Specifically, DCCP provides:

- An unreliable flow of datagrams, with acknowledgements.
- Reliable handshakes 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.
- Congestion control incorporating Explicit Congestion Notification (ECN) and the ECN Nonce, as per [RFC 3168] and [RFC 3540].
- Acknowledgement mechanisms communicating packet loss and ECN mark information. Acks are transmitted as reliably as the relevant congestion control mechanism requires, possibly completely reliably.
- Optional mechanisms that tell the sending application, with high reliability, which data packets reached the receiver, and whether those packets were ECN marked, corrupted, or dropped in the receive buffer.
- Path Maximum Transfer Unit (PMTU) discovery, as per [RFC 1191].

DCCP is intended for applications, such as streaming media and Internet telephony, where reliable in-order delivery, combined with congestion control, can result in some information arriving at the receiver after it is no longer of use. So far, most such applications have either used TCP, with the attendant quality problems caused by late data delivery, or used UDP and implemented their own congestion control (or no congestion control at all). DCCP provides standard congestion control mechanisms for such applications. It enables 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.

DCCP's target applications require the flow-based semantics of TCP, but do not want TCP's in-order delivery and reliability, or would like different congestion control dynamics than TCP.

2. Design Rationale

DCCP was intended to be used by applications that currently use UDP without end-to-end congestion control. Most streaming UDP applications should have little reason not to switch to DCCP, once it is deployed. Thus, DCCP was designed to have as little overhead as possible, both in terms of the packet header size and in terms of the state and CPU overhead required at end hosts. Only the minimal necessary functionality was included in DCCP, leaving other functionality, such as forward error correction (FEC), semi-reliability, and multiple streams, to be layered on top of DCCP as desired. This desire for minimal overhead is also one of the reasons to avoid proposing an unreliable variant of the Stream Control Transmission Protocol (SCTP, [RFC 2960]).

Different forms of conformant congestion control are appropriate for different applications. For example, applications such as on-line games might want to make quick use of any available bandwidth. Other applications, such as streaming media, might trade off this responsiveness for a steadier, less bursty rate, since sudden rate changes cause unacceptable UI glitches (such as audible pauses or clicks in the playout stream). Thus, DCCP allows 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, [RFC 3448]), a form of equation-based congestion control, minimizes abrupt changes in the sending rate while maintaining longer-term fairness with TCP.

DCCP also lets unreliable traffic safely use ECN. 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.

We chose not to require the use of the Congestion Manager [RFC 3124], which allows multiple concurrent streams between the same sender and receiver to share congestion control. The current Congestion Manager can only be used by applications that have their own end-to-end feedback about packet losses, but this is not the case for many of the applications currently using UDP. In addition, the current Congestion Manager does not easily support multiple congestion control mechanisms, or 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. DCCP should be able to make use of CM where desired by the application, but we do not see any benefit in making the deployment of DCCP contingent on the deployment of CM itself.

3. Conventions and Terminology

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

3.1. Numbers and Fields

All multi-byte numerical quantities in DCCP, such as port numbers, 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.

Random numbers in DCCP are used for their security properties, and MUST be chosen according to the guidelines in [RFC 1750].

3.2. Parts of a Connection

Each DCCP connection runs between two endpoints, which we often name DCCP A and DCCP B.

DCCP connections are actively initiated by one endpoint. The active endpoint is called the client, and the passive endpoint is called the server.

DCCP connections are bidirectional; data may pass from either endpoint to the other. This means that data and acknowledgements may be flowing in both directions simultaneously.

Logically, however, a DCCP connection consists of two separate unidirectional connections, called half-connections. Each half-connection consists of the data packets sent by one endpoint and the corresponding acknowledgements sent by the other endpoint. We can illustrate this as follows:

```
+--------+  A-to-B half-connection:         +--------+
|        |    -->    data packets    -->    |        |
|        |    <--  acknowledgements  <--    |        |
| DCCP A |                                  | DCCP B |
|        |  B-to-A half-connection:         |        |
|        |    <--    data packets    <--    |        |
+--------+    -->  acknowledgements  -->    +--------+
```

Although they are logically distinct, in practice the half-connections overlap; a DCCP-DataAck packet, for example, contains application data relevant to one half-connection and acknowledgement information relevant to the other.

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

3.3. Features

A feature is a DCCP connection attribute, identified by a feature number and an endpoint, on whose value the two endpoints agree. Many properties of a DCCP connection are controlled by features, including the congestion control mechanisms in use on the two half-connections, whether mobility is allowed, and whether ECN is supported. The endpoints can achieve agreement by out-of-band communication, or through the exchange of feature negotiation options in DCCP headers.

The notation F/A represents the feature with feature number F located at DCCP endpoint A; the feature F/B has the same feature number, but is located at the other endpoint. Both DCCP A and DCCP B know, and agree on, the values of both F/A and F/B, but F/A and F/B may have different values.

DCCP A is called the feature location for all features F/A, and the feature remote for all features F/B.

3.4. Round-Trip Times

We sometimes refer to a round-trip time for setting timers, for example. If no useful round-trip time estimate is available, a DCCP implementation SHOULD use 0.2 seconds instead.

3.5. 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.

4. Overview

DCCP's high-level connection dynamics should seem familiar to anyone who knows TCP. DCCP connections, like TCP connections, progress through three phases: initiation (including a three-way handshake), data transfer, and termination. Data can flow both ways over the connection. An acknowledgement framework lets senders discover how much data has been lost; congestion control uses this information to avoid unfairly congesting the network. Of course, DCCP provides unreliable datagram semantics, not TCP's reliable bytestream semantics. The application must package its data into explicit frames, and must retransmit its own data as necessary. It may be useful to think of DCCP either as TCP minus bytestream semantics and reliability, or as UDP plus congestion control, handshakes, and acknowledgements.

4.1. Packet Types

DCCP uses eleven packet types to implement various protocol functions. For example, every new connection attempt begins with a DCCP-Request packet sent by the client. A DCCP-Request packet thus resembles a TCP SYN; but DCCP-Request is a packet type, not a flag, so there's no way to send an unexpected combination such as TCP's SYN+FIN+ACK+RST.

Eight packet types occur during the progress of a typical connection---two only during the initiation phase, three during the data transfer phase, and three only during the termination phase:

```
Client                                            Server
------                                            ------
                 (1) Initiation
DCCP-Request -->
                                      <-- DCCP-Response
DCCP-Ack -->
                 (2) Data transfer
DCCP-Data, DCCP-Ack, DCCP-DataAck -->
             <-- DCCP-Data, DCCP-Ack, DCCP-DataAck
                 (3) Termination
                                      <-- DCCP-CloseReq
DCCP-Close -->
                                         <-- DCCP-Reset
```

Note the three-way handshakes during initiation and termination. The three remaining packet types are used for special purposes: when an endpoint moves, or to resynchronize after bursts of loss.

Every DCCP packet starts with a common, 12-byte generic header, but different packet types may include different amounts of additional data. For example, the DCCP-Ack packet type includes an Acknowledgement Number. Every packet type may also contain options, up to around 1000 bytes' worth.

All of the packet types are described below.

DCCP-Request
: Sent by the client to initiate a connection (the first part of the three-way handshake).

DCCP-Response
: Sent by the server in response to a DCCP-Request (the second part of the three-way handshake).

DCCP-Data
: Used to transmit data.

DCCP-Ack
: Used for pure acknowledgements.

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

DCCP-CloseReq
Sent by the server to request that the client close the connection.

DCCP-Close
Used to close the connection; elicits a DCCP-Reset in response.

DCCP-Reset
Used to terminate the connection, either normally or abnormally.

DCCP-Move
Supports multihoming and mobility.

DCCP-Sync, **DCCP-SyncAck**
Used to resynchronize sequence numbers after large bursts of loss.

4.2. Sequence Numbers

Each DCCP packet carries a sequence number, so that losses can be detected and reported. But unlike TCP's byte-based sequence numbers, DCCP sequence numbers are attached to packets. Each packet sent increments the sequence number by one. For example:

```
DCCP A                                           DCCP B
------                                           ------
DCCP-Data(seqno 1) -->
DCCP-Data(seqno 2) -->
                   <-- DCCP-Ack(seqno 10, ackno 2)
DCCP-DataAck(seqno 3, ackno 10) -->
                           <-- DCCP-Data(seqno 11)
```

Note that even DCCP-Ack pure acknowledgements increment the sequence number; after the DCCP-Ack with sequence number 10, the following DCCP-Data packet uses the next sequence number, 11. This lets the endpoints tell when acknowledgements are lost in the network. It also means that endpoints can get out of sync after a long burst of loss. The DCCP-Sync and DCCP-SyncAck packet types let DCCP recover from large loss bursts; see Section 7.5.

Also note that, since DCCP is an unreliable protocol, there are no retransmissions, and it doesn't make sense to have a cumulative acknowledgement field. Acknowledgement Number (ackno) fields equal the largest sequence number received, rather than the TCP-style smallest sequence number not received. Separate options indicate any intermediate sequence numbers that weren't received.

4.3. States

DCCP endpoints progress through different states during the course of a connection, corresponding roughly to the three phases of initiation, data transfer, and termination. The figure below shows the typical progress through these states for a client and server.

```
Client                                                    Server
------                                                    ------
                  (0) No connection
CLOSED                                                    LISTEN

                  (1) Initiation
REQUEST      DCCP-Request -->
                             <-- DCCP-Response            RESPOND
PARTOPEN     DCCP-Ack or DCCP-DataAck -->

                  (2) Data transfer
OPEN          <-- DCCP-Data, Ack, DataAck -->             OPEN

                  (3) Termination
                             <-- DCCP-CloseReq            CLOSEREQ
CLOSING      DCCP-Close -->
                                <-- DCCP-Reset            CLOSED
TIMEWAIT
CLOSED
  The client and server's typical progress through states.
```

The states are as follows; Section 8 describes them in more detail.

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.

PARTOPEN
The client socket enters this state, from REQUEST, after receiving a DCCP-Response from the server. This state represents the third phase of the three-way handshake. The client may send data in this state, but it MUST include an Acknowledgement Number on all of its packets.

OPEN
The central, data transfer portion of a DCCP connection. Client and server enter into this state from PARTOPEN 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 TIMEWAIT state.

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

TIMEWAIT
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. One MSL, or Maximum Segment Lifetime, is the maximum length of time a packet could survive in the network.

4.4. Congestion Control

DCCP connections are congestion controlled. Unlike TCP, however, DCCP supports multiple congestion control mechanisms for applications to choose from. In fact, the two half-connections can be governed by different mechanisms. Each mechanism corresponds to a one-byte congestion control identifier, or CCID. A CCID 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 during connection initiation. So far, CCIDs 2 and 3 have been defined for use with DCCP; CCID 0 is reserved, and CCID 1 is used for special purposes (see Section 10.1).

CCID 2 corresponds to TCP-like Congestion Control, which is 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 indicate congestion; the response to congestion is to halve the congestion window. Acknowledgements in CCID 2 contain the sequence numbers of all received packets within some window, similar to a super selective-acknowledgement (SACK, [RFC 3517]).

CCID 3 provides TFRC Congestion Control, 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 containing 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.

The behaviors of CCIDs 2 and 3 are fully defined in separate profile documents [CCID 2 PROFILE] [CCID 3 PROFILE].

4.5. Features

Agreement on DCCP feature values is achieved by explicit negotiation, using options in DCCP packet headers. This generally happens at connection startup, but negotiation can begin at any time. The relevant options are Change L, Confirm L, Change R, and Confirm R, with the "L" options sent by the feature location and the "R" options sent by the feature remote.

A Change R message says to the peer, "change this feature value on your side". The peer responds with a Confirm L, meaning "I've changed it". The suggested option setting in Change R can sometimes contain multiple values, which are sorted in preference order. For example:

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

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

In the second exchange, the client requests that the server use either CCID 3 or CCID 4, with 3 preferred. The server chooses 4, giving its preference list of "4 2".

A party that wants to change a feature located at itself issues a "Change L" option, which elicits a "Confirm R" in reply.

```
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 describes these options further.

4.6. Other Differences from TCP

Interesting differences between DCCP and TCP, apart from those discussed so far, include:

- **Copious space for options** (up to 1020 bytes).
- **Different acknowledgement formats.** The CCID for a connection determines how much ack information needs to be transmitted. In CCID 2 (TCP-like), this is about one ack per 2 packets, and each ack must declare exactly which packets were received; in CCID 3 (TFRC), it's about one ack per RTT, and acks must declare at minimum just the lengths of recent loss intervals.
- **Denial-of-service (DoS) protection.** Several DCCP mechanisms attempt to let servers limit the amount of state possibly-misbehaving clients can force them to maintain. An Init Cookie option, analogous to TCP's SYN Cookies [SYNCOOKIES], avoids SYN-flood-like attacks. Only one connection endpoint need hold TIMEWAIT state; the DCCP-CloseReq packet, which may only be sent by the server, passes that state to the client. Various rate limits let servers avoid attacks that might force extensive computation or packet generation.
- **Distinguishing different kinds of loss.** A Data Dropped option (Section 11.7) lets an endpoint declare that a packet was dropped because of corruption, because of receive buffer overflow, and so on. This facilitates research into more appropriate rate-control responses for these non-network-congestion losses (although currently all losses will cause a congestion response).
- **Acknowledgement readiness.** In TCP, a packet is acknowledged only when the data is queued for delivery to the application. This does not make sense in DCCP, where an application might request a drop-from-front receive buffer, for example. We

acknowledge a packet when its options have been processed. The Data Dropped option may later say that the packet's payload was discarded.

- **Integrated support for mobility and multihoming** via the DCCP-Move packet type.
- **No receive window.** DCCP is a congestion control protocol, not a flow control protocol.
- **No simultaneous open.** Every connection has one client and one server.
- **No half-closed states.** DCCP has no states corresponding to TCP's FINWAIT and CLOSEWAIT, where one half-connection is explicitly closed while the other is still active.

4.7. Example Connection

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

```
    Client                                            Server
    ------                                            ------
0.  [CLOSED]                                        [LISTEN]
1.  DCCP-Request -->
2.                                    <-- DCCP-Response
3.  DCCP-Ack -->
                                           <-- DCCP-Ack
4.  DCCP-Data, DCCP-Ack, DCCP-DataAck -->
             <-- DCCP-Data, DCCP-Ack, DCCP-DataAck
5.                                    <-- DCCP-CloseReq
6.  DCCP-Close -->
7.                                       <-- DCCP-Reset
8.  [TIMEWAIT]
```

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. There might follow 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.
4. 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.

5. The server sends a DCCP-CloseReq packet requesting a close.
6. The client sends a DCCP-Close packet acknowledging the close.
7. The server sends a DCCP-Reset packet with Reset Code 1, "Closed", and clears its connection state. In DCCP, unlike TCP, Resets are part of normal connection termination; see Section 5.6.
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.

An alternative connection closedown sequence is initiated by the client:

5b. The client sends a DCCP-Close packet closing the connection.

6b. The server sends a DCCP-Reset packet with Reset Code 1, "Closed", and clears its connection state.

7b. 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.

5. Header Formats

The variable-length DCCP header appears first in every DCCP packet. A header can be from 12 to 1020 bytes long. The initial 12 bytes of the header are the same regardless of packet type. Following this comes optional additional fixed-length fields, depending on the packet type, and then a variable-length list of options. Finally, some packet types include application data.

```
+---------------------------------------+  -.
|            Generic Header             |   |
+---------------------------------------+   |
| Additional Fields (depending on type) |   +- DCCP Header
+---------------------------------------+   |
|          Options (optional)           |   |
+=======================================+  -'
|      Application Data (optional)      |
+=======================================+
```

5.1. Generic Header

The DCCP generic header generally takes 12 bytes.

```
 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| Res |              Sequence Number                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Actually, there are two types of generic header, depending on the value of X, the Extended Sequence Numbers bit. If X is zero, the Sequence Number field takes 24 bits, as above. If X is one, the Sequence Number field extends for an additional 24 bits, for a total of 48:

```
 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| Res |          Sequence Number (high bits)          .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
.          Sequence Number (low bits)           |  Reserved   |T|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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. Source Ports SHOULD be chosen randomly, to reduce the likelihood of attack.

Data Offset: 8 bits

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

CCVal: 4 bits

Used by the HC-Sender CCID. For example, the A-to-B CCID's sender, which is active at DCCP A, MAY send 4 bits of information per packet to its receiver by encoding that information in CCVal. CCVal MUST be set to zero unless the HC-Sender CCID specifies a different value.

Checksum Coverage (CsCov): 4 bits

Checksum Coverage specifies what parts of the packet are covered by the Checksum field. This always includes the DCCP header and options, but if applications request it, some or all of the application data may be excluded. This can improve performance on noisy links, assuming the application can tolerate corruption. See Section 9.

Checksum: 16 bits

The Internet checksum of the packet's DCCP header (including options), a network-layer pseudoheader, and, depending on Checksum Coverage, some or all of the application data. See Section 9.

Type: 4 bits

The Type field specifies the type of the packet. The following values are defined:

```
Type   Meaning
----   -------
  0    DCCP-Request
  1    DCCP-Response
  2    DCCP-Data
  3    DCCP-Ack
  4    DCCP-DataAck
  5    DCCP-CloseReq
  6    DCCP-Close
  7    DCCP-Reset
  8    DCCP-Move
  9    DCCP-Sync
 10    DCCP-SyncAck
11-15  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. Very-high-rate connections SHOULD set X to one, and use 48-bit sequence numbers, to gain increased protection against wrapped sequence numbers and attacks. See Section 7.6.

Reserved (Res): 3 bits
The version of DCCP specified here MUST ignore this field on received packets, and MUST set it to all zeroes on generated packets.

Sequence Number: 24 or 48 bits
Identifies the packet uniquely in the sequence of all packets the source sent on this connection. Sequence Number increases by one with every packet sent, including packets such as DCCP-Ack that carry no application data. See Section 7.

Sequence Number Transition (T): 1 bit [X=1 only]
Set to one to indicate an ongoing transition from 24-bit to 48-bit sequence numbers. See Section 7.6.

Many packet types also carry an Acknowledgement Number in the four or eight bytes immediately following the generic header. When X=0, its format is:

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

And when X=1:

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

Acknowledgement Number: 24 or 48 bits
The Acknowledgement Number field generally 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.

Reserved: 8 bits

The version of DCCP specified here MUST ignore these fields on received packets, and MUST set them to all zeroes on generated packets.

5.2. DCCP-Request Header

A client initiates a DCCP connection by sending a DCCP-Request 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=0 (DCCP-Request)                  /
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                         Service Code                          |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                      Options                  /    Padding     |
 +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
 |                       Application Data                        |
 |                              ...                              |
```

Service Code: 32 bits

Describes the service to which the client application wants to connect. Examples might include RTSP and DOOM. Service Codes are intended to make application protocols independent of well-known ports, and help middleboxes identify the protocol used on a given connection. See Section 8.1.2.

5.3. DCCP-Response Header

The server responds to valid DCCP-Request packets with DCCP-Response packets. This is the second phase of the three-way handshake.

```
  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            |
(+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+) when
(.       Acknowledgement Number (low bits)       |    Reserved    |) X=1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                         Service Code                          |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                      Options                  /    Padding     |
 +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
 |                       Application Data                        |
 |                              ...                              |
```

Acknowledgement Number: 24 or 48 bits

The Acknowledgement Number field will generally equal the Sequence Number from the DCCP-Request.

Service Code: 32 bits
Echoes the Service Code on the DCCP-Request.

5.4. DCCP-Data, DCCP-Ack, and DCCP-DataAck Headers

The central data transfer portion of every DCCP connection uses DCCP-Data, DCCP-Ack, and DCCP-DataAck packets. DCCP-Data packets carry application data.

```
  0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 /              Generic DCCP Header (12 or 16 bytes)             /
 /                    with Type=2 (DCCP-Data)                    /
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                     Options                   /    Padding    |
 +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
 |                        Application Data                       |
 |                              ...                              |
```

DCCP-Ack packets dispense with the data, but contain an Acknowledgement Number. They are used for pure acknowledgements.

```
  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            |
(+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+)when
(.       Acknowledgement Number (low bits)       |   Reserved    |)X=1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                     Options                   /    Padding    |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

DCCP-DataAck packets carry both application 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            |
(+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+)when
(.       Acknowledgement Number (low bits)       |   Reserved    |)X=1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                     Options                   /    Padding    |
 +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
 |                        Application Data                       |
 |                              ...                              |
```

DCCP-Data and DCCP-DataAck packets may contain zero application data bytes if the application sends a zero-length datagram. Also, a DCCP-Ack packet need not have a zero-length application data area. The receiver MUST ignore any "application data" in a DCCP-Ack packet. The sender will not generally send such data, but it may occasionally do so---to perform PMTU discovery without risking loss of user data, for example.

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

5.5. DCCP-CloseReq and DCCP-Close Headers

DCCP-CloseReq and DCCP-Close packets begin the handshake that normally terminates a connection. Either client or server may send a DCCP-Close packet, which will elicit a DCCP-Reset packet (see the next section). Only the server can send a DCCP-CloseReq packet, which indicates that the server wants to close the connection, but does not want to hold its TIMEWAIT state.

```
  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 (DCCP-CloseReq) or 6 (DCCP-Close)        /
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |   Reserved    |             Acknowledgement Number            |
(+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+) when
(.       Acknowledgement Number (low bits)       |   Reserved    |) X=1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                         Options               /    Padding    |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The receiver MUST ignore any "application data" in a DCCP-CloseReq or DCCP-Close packet.

5.6. DCCP-Reset Header

DCCP-Reset packets unconditionally shut down a connection. Connections normally terminate 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.

```
  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             |
(+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+)when
(.       Acknowledgement Number (low bits)       |   Reserved    |)X=1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |  Reset Code   |    Data 1     |    Data 2     |    Data 3     |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                     Options                   /    Padding    |
 +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
 |                          Error Text                           |
 |                              ...                              |
```

Reset Code: 8 bits
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.

Error Text (application data area)
If present, Error Text is a human-readable text string, preferably in English and encoded in Unicode UTF-8, that describes the error in more detail. For example, a DCCP-Reset with Reset Code 12, "Aggression Penalty", might contain Error Text such as "Aggression Penalty: Received 3 bad ECN Nonce Echoes, assuming misbehavior".

The following Reset Codes are currently defined. The "Data" columns describe what the Data fields contain for a given Code. N/A means the Data field MUST be set to 0 by the sender of the DCCP-Reset and ignored by its receiver.

Reset Code	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	8.3
2	Aborted	N/A	N/A	N/A	8.1.1
3	No Connection	N/A	N/A	N/A	8.3.1
4	Packet Error	packet type	N/A	N/A	8.3.1
5	Option Error	option number	option data (if any)		
6	Mandatory Error	option number	option data (if any)		5.9.2
7	Extended Seqnos	N/A	N/A	N/A	7.6
8	Connection Refused	N/A	N/A	N/A	8.1.3
9	Bad Service Code	N/A	N/A	N/A	8.1.3
10	Too Busy	N/A	N/A	N/A	8.1.3
11	Bad Init Cookie	N/A	N/A	N/A	8.1.4
12	Aggression Penalty	N/A	N/A	N/A	12.2
13	Move Refused	N/A	N/A	N/A	14.4
13-127	Reserved				
128-255	CCID-specific codes		... variable ...		10.4

5.7. DCCP-Move Header

The DCCP-Move packet type is part of DCCP's support for multihoming and mobility, which is described further in Section 14. 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 a new address and port number, which are read off the packet's network header and generic DCCP header. The old address and port are defined through a Mobility ID, which provides some protection against hijacked connections.

```
  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            |
(+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+)when
(.       Acknowledgement Number (low bits)       |   Reserved    |)X=1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                    Mobility ID (high bits)                    .
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 .                   Mobility ID (bits 64-95)                    .
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 .                   Mobility ID (bits 32-63)                    .
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 .                    Mobility ID (low bits)                     |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                     Options                   /    Padding    |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Mobility ID: 128 bits

The value of the receiver's Mobility ID feature. This value uniquely identifies the current connection among the set of connections terminating at the receiver (meaning, the stationary endpoint); it MUST have been set in an earlier exchange. See Section 14.2.

The receiver MUST ignore any "application data" in a DCCP-Move packet.

5.8. DCCP-Sync and DCCP-SyncAck Headers

DCCP-Sync packets help DCCP endpoints recover synchronization after bursts of loss, or recover from half-open connections. Each valid DCCP-Sync received immediately elicits a DCCP-SyncAck.

```
  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) or 10 (DCCP-SyncAck)         /
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |   Reserved    |             Acknowledgement Number            |
(+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+)when
(.       Acknowledgement Number (low bits)       |   Reserved    |)X=1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                     Options                   /    Padding    |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The Acknowledgement Number on DCCP-Sync and DCCP-SyncAck packets need not equal the generating endpoint's greatest valid sequence number received (GSR). This differs from Acknowledgement Numbers on all other packet types. If a DCCP-Sync was generated in response to a packet with invalid sequence numbers, then the DCCP-Sync's Acknowledgement Number will equal the invalid packet's sequence number. The

Acknowledgement Number on any DCCP-SyncAck packet MUST correspond to a received, valid DCCP-Sync's Sequence Number; in the presence of reordering, this might not equal GSR.

The receiver MUST ignore any "application data" in a DCCP-Sync or DCCP-SyncAck packet.

5.9. Options

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 first option in the packet header.

The following options are currently defined:

```
        Option                               Section
 Type   Length     Meaning                  Reference
 ----   ------     -------                  ---------
   0       1       Padding                    5.9.1
   1       1       Mandatory                  5.9.2
   2       1       Slow Receiver              11.6
 3-31      1       Reserved
  32    variable   Change L                   6.1
  33    variable   Confirm L                  6.2
  34    variable   Change R                   6.1
  35    variable   Confirm R                  6.2
  36    variable   Init Cookie                8.1.4
  37      4-5      NDP Count                  7.7
  38    variable   Ack Vector [Nonce 0]       11.4
  39    variable   Ack Vector [Nonce 1]       11.4
  40    variable   Data Dropped               11.7
  41       6       Timestamp                  13.1
  42      6-10     Timestamp Echo             13.3
  43      4-6      Elapsed Time               13.2
  44       4       Data Checksum              9.3
 45-127 variable   Reserved
128-255 variable   CCID-specific options      10.4
```

This section describes two generic options, Padding and Mandatory. Other options are described later.

5.9.1. Padding Option

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

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

5.9.2. 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, generally using Reset Code 6, "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 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. Section 6.6.8 describes the behavior of Mandatory feature negotiation options in more detail.

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

6. Feature Negotiation

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

All these options have the same format. The first byte of option data is the feature number, and the second and subsequent data bytes hold one or more feature values. The feature values are generally arranged in a linear preference list, where the first value is most preferred.

```
+--------+--------+--------+--------+--------
|  Type  | Length |Feature#| Value(s) ...
+--------+--------+--------+--------+--------
```

Together, the feature number and the option type ("L" or "R") uniquely identify the feature to which an option applies. The exact format of the Value(s) area depends on the feature number.

6.1. Change Options

Change L and Change R options initiate feature negotiation. Either endpoint can start a negotiation for any feature; if DCCP A wants to start a negotiation for feature F/A, it will send a Change L option, while to start a negotiation for F/B, it will send a Change R option. Change options are retransmitted until some response is received. Normal Change options contain at least one Value, and thus have length at least 4.

```
           +--------+--------+--------+--------+--------
Change L:  |00100000| Length |Feature#| Value(s) ...
           +--------+--------+--------+--------+--------
            Type=32

           +--------+--------+--------+--------+--------
Change R:  |00100010| Length |Feature#| Value(s) ...
           +--------+--------+--------+--------+--------
            Type=34
```

The endpoint may check a feature's current value without attempting to change it by sending an empty Change option, containing just the feature number. Such options have length 3. The endpoints must agree on feature values anyway, so these options are useful in practice only in special situations, such as when a middlebox introduced in the middle of a connection wants to check a feature value.

6.2. Confirm Options

Confirm L and Confirm R options complete feature negotiation, and are sent in response to Change R and Change L options, respectively. Confirm options MUST NOT be generated except in response to Change options. Confirm options need not be retransmitted, since Change options are retransmitted as necessary. Normal Confirm options contain the selected Value, possibly followed by the sender's preference list.

```
           +--------+--------+--------+--------+--------
Confirm L: |00100001| Length |Feature#| Value(s) ...
           +--------+--------+--------+--------+--------
            Type=33

           +--------+--------+--------+--------+--------
Confirm R: |00100011| Length |Feature#| Value(s) ...
           +--------+--------+--------+--------+--------
            Type=35
```

If an endpoint receives an invalid Change option -- with an unknown feature number, or an invalid value -- it will respond with an empty Confirm option containing no value. Such options have length 3.

6.3. Reconciliation Rules

Reconciliation rules determine how the two sets of preferences for a given feature are resolved into a unique result. The reconciliation rule depends only on the feature number. Each reconciliation rule must have the property that the result is uniquely determined given the contents of Change options sent by the two endpoints.

All current DCCP features use one of two reconciliation rules, server-priority ("SP") and non-negotiable ("NN").

6.3.1. Server-Priority

The feature value is a fixed-length byte string (length determined by the feature number). Each Change option contains a preference list of values, with the most preferred value coming first. Each Confirm option contains the confirmed value, followed by the confirmer's preference list. Thus, the feature's current value will generally appear twice in

Confirm options' data, once as the current value and once in the confirmer's preference list. Even responses to empty Change options contain the whole preference list.

To reconcile the 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 feature's value MUST NOT change, and the Confirm option will confirm the feature's previous value (unless the Change option was Mandatory; see Section 6.6.8).

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, the preference lists MUST remain stable until the negotiation has closed.

6.3.2. Non-Negotiable

The feature value is a byte string. Each option contains exactly one feature value. The feature location signals a value change by sending Change L options. The feature remote MUST accept any valid value, responding with a Confirm R option containing the new value, and it MUST send empty Confirm R options in response to invalid values. Non-negotiable features aren't really negotiated; they use feature negotiation as a mechanism for achieving reliability. Change R and Confirm L options MUST NOT be sent for non-negotiable features.

6.4. Feature Numbers

This document defines the following feature numbers.

```
                                           Rec'n Initial        Section
Number   Meaning                           Rule   Value  Req'd Reference
------   -------                           -----  -----  ----- ---------
   0     Reserved
   1     Congestion Control ID (CCID)       SP      2      Y      10
   2     ECN Capable                        SP      1      Y      12.1
   3     Sequence Window                    NN     100     Y      7.5.4
   4     Sequence Transition Capable        SP      0      N      7.6.4
   5     Mobility Capable                   SP      0      N      14.1
   6     Mobility ID                        NN      0      N      14.2
   7     Ack Ratio                          NN      2      N      11.3
   8     Send Ack Vector                    SP      0      N      11.5
   9     Send NDP Count                     SP      0      N      7.7.2
  10     Check Data Checksum                SP      0      N      9.3.1
 11-127  Reserved
128-255  CCID-specific features              ?      ?      ?      10.4
```

Rec'n Rule The reconciliation rule used for the feature. SP is server-priority and NN is non-negotiable.

Initial Value The initial value for the feature. Every feature has a known initial value.

Req'd This column is "Y" iff every DCCP implementation MUST understand the feature. If it is "N", then the feature behaves like an extension (see Section 16), and it is safe to respond to Change options for the feature with empty Confirm options. Of course, a CCID might require the feature; a DCCP that implements CCID 2 MUST support Ack Ratio and Send Ack Vector, for example.

6.5. Examples

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

```
           Client                          Server
 1. Change R(CCID, 2 3 1)  -->
    ("2 3 1" is client's value preference list)
 2.                        <--  Confirm L(CCID, 3, 3 2 1)
                          (3 is the negotiated value;
                          "3 2 1" is server's pref list)
            * agreement that CCID/Server = 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/Server = 3 *

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

This example shows a simultaneous negotiation.

```
           Client                          Server
1a. Change R(CCID, 2 3 1)  -->
 b.                        <--  Change L(CCID, 3 2 1)
            (both endpoints in CHANGING)
2a.                        <--  Confirm L(CCID, 3, 3 2 1)
 b. Confirm R(CCID, 3, 2 3 1)  -->
            (both endpoints in STABLE)
            * agreement that CCID/Server = 3 *
```

Example Change and Confirm options follow, with their byte encodings. Each option is sent by DCCP A.

Change L(CCID, 2 3) = 32,5,1,2,3
: I want to change CCID/A's value (feature number 1, a server-priority feature); my preferred values are 2 and 3, in that preference order.

Change L(Sequence Window, 1024) = 32,6,3,0,4,0
: Change Sequence Window/A's value (feature number 3, a non-negotiable feature) to the 3-byte string 0,4,0 (the value 1024).

Empty Change L(CCID) = 32,3,1
: Tell me CCID/A's value using a Confirm R option.

Confirm L(CCID, 2, 2 3) = 33,6,1,2,2,3
: I've changed CCID/A's value to 2; my preferred values are 2 and 3, in that preference order.

Empty Confirm L(126) = 33,3,126
: I don't implement feature number 126, or your proposed value for feature 126/A was invalid.

Change R(CCID, 3 2) = 34,5,1,3,2
Please change CCID/B's value; my preferred values are 3 and 2, in that preference order.

Empty Change R(CCID) = 34,3,1
Tell me CCID/B's value using a Confirm L option.

Confirm R(CCID, 2, 3 2) = 35,6,1,2,3,2
I've changed CCID/B's value to 2; my preferred values were 3 and 2, in that preference order.

Confirm R(Sequence Window, 1024) = 35,6,3,0,4,0
I've changed Sequence Window/B's value to the 3-byte string 0,4,0 (the value 1024).

Empty Confirm R(126) = 35,3,126
I don't implement feature number 126, or your proposed value for feature 126/B was invalid.

6.6. Option Exchange

A few basic rules govern feature negotiation option exchange.

1. Every non-reordered Change option gets a Confirm option in response.
2. Change options are retransmitted until some response is received.
3. Preference lists don't change during a negotiation.
4. Feature negotiation options are processed in strictly increasing order by Sequence Number.

The rest of this section describes the consequences of these rules in more detail.

6.6.1. Normal Exchange

Change options are generated when a DCCP endpoint wants to change the value of some feature. Generally, this will happen at the beginning of a connection, although it may happen at any time. We say the endpoint "generates" or "sends" a Change L or Change R option; but, of course, the option must be attached to a packet. The endpoint may attach the option to a packet it would have generated anyway (such as a DCCP-Request), or it may create a new packet just to carry the options (often a DCCP-Sync). If it does create a new packet, it MUST NOT create more than one such packet per round-trip time (or 0.2 seconds, if no RTT is available).

On receiving a Change L or Change R option, a DCCP endpoint examines the included preference list, reconciles that with its own preference list, calculates the new value, and sends back a Confirm R or Confirm L option, respectively, informing its partner of the new value. The rule for reconciling the two preference lists is feature-specific; see Section 6.3. Every non-reordered Change option MUST result in a corresponding Confirm option. Any packet including a Confirm option MUST carry an Acknowledgement Number; thus, Confirm options are not allowed on DCCP-Request and DCCP-Data packets. Again, generated Confirm options may be attached to packets that would have been sent anyway (such as DCCP-Response or DCCP-SyncAck), or to new packets (usually DCCP-Ack).

The Change-sending endpoint MUST wait to receive a corresponding Confirm option before changing its stored feature value. The Confirm-sending endpoint changes its stored feature value as soon as it sends the Confirm.

DCCP endpoints effectively exist in one of two states, STABLE and CHANGING, relative to each feature. STABLE is the normal state, where the endpoint knows the feature's value and thinks the other endpoint agrees. An endpoint enters the CHANGING state when it first sends a Change for the feature, and returns to STABLE once it receives a corresponding Confirm.

6.6.2. Loss and Retransmission

Packets containing Change and Confirm options might be lost or delayed by the network. Therefore, Change options are retransmitted to achieve reliability.

A CHANGING endpoint retransmits a Change option once it realizes that it has not heard back from the other endpoint. Each retransmitted Change option MUST contain exactly the same payload as the original. The endpoint may piggyback its Change options on packets it would have sent anyway. If it generates new packets for feature negotiation, it MUST use an exponential-backoff timer. The timer's initial value is set to approximately one or two round-trip times (or 0.2-0.4 seconds, if no RTT is available), and it is pinned at roughly 32 RTTs.

A CHANGING endpoint MUST continue retransmitting Change options until it gets some response. Its only recourse is to reset the connection, which it SHOULD NOT do until at least 12 transmissions have failed.

Change options SHOULD NOT be transmitted more frequently than once per RTT, or the reordering protection below would prevent any Confirm option from being accepted (since no Confirm would acknowledge the most recently transmitted Change).

Confirm options are never retransmitted, but the Confirm-sending endpoint MUST generate a new Confirm option for every non-reordered Change it receives.

6.6.3. Reordering

Reordering might cause packets containing Change and Confirm options to arrive in an unexpected order. Endpoints MUST be robust to reordering, by ignoring feature negotiation options that do not arrive in strictly-increasing order by Sequence Number.

The most straightforward way to implement this requirement is for an endpoint to associate two sequence number variables with every feature F/X, as follows.

F/X.GSR The Greatest Sequence Number Received from the other endpoint on a packet containing a Change or Confirm option for feature F/X.

F/X.GSS The Greatest Sequence Number Sent by this endpoint on a packet containing a Change option for feature F/X.

Then DCCP A will check options relating to feature F/A as follows:

1. Ignore any received Change R(F) option whose packet's Sequence Number is not greater than F/A.GSR.

2. Ignore any received Confirm R(F) option whose packet's Sequence Number is not greater than F/A.GSR, or whose packet could not have acknowledged F/A.GSS. Specifically, if the Acknowledgement Number is less than F/A.GSS, the endpoint MUST ignore the Confirm; and if the packet has an Ack Vector indicating that F/A.GSS was not received, the endpoint MAY ignore the Confirm.

A similar procedure applies options relating to feature F/B, namely Change L(F) and Confirm L(F), except that F/B.GSR and F/B.GSS are checked.

A less state-intensive way to implement this requirement would be to share the F.GSR and F.GSS variables among all features, rather than keeping one pair per feature. Then the feature negotiation options on any received packet would be treated as a unit (either all accepted or all rejected).

Checking Confirm options is easier if the endpoint only sends Change options on packet types that will be acknowledged immediately, namely DCCP-Request, DCCP-Response, and DCCP-Sync. Then there is never any need to check Ack Vectors, although checking Ack Vectors is NOT MANDATORY anyway.

6.6.4. Preference Changes

Endpoints MUST NOT change their preference lists in the middle of a negotiation. This is because, if a preference list changed in the middle of a negotiation and the right packets were lost, the negotiation could terminate with the endpoints thinking the feature had different values. In particular, an endpoint MUST NOT change its preference list while in the CHANGING state; this ensures that every Change option sent during that negotiation will contain the same data.

6.6.5. Simultaneous Negotiation

The two endpoints might simultaneously open negotiation for the same feature, after which an endpoint in the CHANGING state will receive a Change option for the same feature. Such received Change options can act as responses to the original Change options. The CHANGING endpoint MUST examine the received Change's preference list, reconcile that with its own preference list (as expressed in its generated Change options), and generate the corresponding Confirm option. It can then transition to the STABLE state.

6.6.6. Unknown Features

An endpoint may receive a Change option referring to some feature number it does not understand. This is particularly likely to happen when an extended DCCP converses with a non-extended DCCP. The receiving endpoint MUST respond to such Change options with corresponding empty Confirm options (that is, Confirm options containing no data), which inform the CHANGING endpoint that the feature was not understood. However, if the Change option was preceded by a Mandatory option, the connection MUST be reset; see Section 6.6.8.

On receiving an empty Confirm option for some feature, the CHANGING endpoint MUST transition back to the STABLE state, leaving the feature's value unchanged. Section 16 suggests that the default value for any extension feature should correspond to "extension not available".

An endpoint will also send an empty Confirm option when it understood the Change's feature number, but considered the Change's value invalid or inappropriate for the feature. The next section describes this further.

Some features are required to be understood by all DCCPs (see Section 6.4); the CHANGING endpoint SHOULD reset the connection (with Reset Code 5, "Option Error") if it receives an empty Confirm option for such a feature.

Since Confirm options are generated only in response to Change options, an endpoint should never receive a Confirm option referring to a feature number it does not understand. Endpoints MUST either reset the connection on receiving such options, or just ignore the options.

6.6.7. Invalid Options

A DCCP endpoint might receive a Change or Confirm option that lists one or more values that it does not understand. Some, but not all, such options are invalid, depending on the relevant reconciliation rule (Section 6.3). For instance:

- All features have length limitiations, and options with invalid lengths are invalid. For example, the Mobility ID feature takes 128-bit values, so valid "Confirm R(Mobility ID)" options have option length 19.
- Some non-negotiable features have value limitations. The Ack Ratio feature takes two-byte, non-zero integer values, so a "Change L(Ack Ratio, 0)" option is never valid. Note that server-priority features do not have value limitations, since unknown values are handled as a matter of course.
- Any Confirm option that selects the wrong value, based on the two preference lists and the relevant reconciliation rule, is invalid.

An endpoint receiving an invalid Change option MUST respond with the corresponding empty Confirm option. An endpoint receiving an invalid Confirm option MUST reset the connection, with Reset Code 5, "Option Error".

6.6.8. Mandatory Feature Negotiation

Change options may be preceded by Mandatory options (Section 5.9.2). Mandatory Change options are processed like normal Change options, except that various failure cases will cause the receiver to reset the connection with Reset Code 6, "Mandatory Failure", rather than send a Confirm option. Specifically, the connection MUST be reset if:

- The Change option's feature number was not understood;
- The Change option's value was invalid, and the receiver would normally have sent an empty Confirm option in response; or
- For server-priority features, there was no shared entry in the two endpoints' preference lists.

There's no reason to mark Confirm options as Mandatory in this version of DCCP, since Confirm options are sent only in response to Change options and therefore can't mention potentially-invalid values or unexpected feature numbers.

6.6.9. Out-of-Band Agreement

An endpoint MUST NOT unilaterally change the value of any DCCP feature. However, endpoints MAY cooperatively change DCCP feature values without using in-band feature negotiation options---by using a separate signalling channel, for example.

6.6.10. State Diagram

This diagram illustrates feature-related state transitions, ignoring sequence number and option validity issues, for the endpoint that is the feature location. For a feature remote state transition diagram, switch the "L"s and "R"s.

```
rcv Confirm R        app/protocol evt : snd Change L
: ignore      +---------------------------------------------+
     +----+   |                                             |
     |    v   |                    rcv Change R             v
  +-------------+ rcv Confirm R    : calc new value, +-------------+
  |             | : accept value     snd Confirm L   |             |
  |   STABLE    |<-----------------------------------|  CHANGING   |
  |             |        rcv empty Confirm R         |             |
  +-------------+        : revert to old value       +-------------+
      |    ^                                             |    ^
      +----+                                             +----+
rcv Change R                                      timeout/rcv non-ack
: calc new value, snd Confirm L                   : snd Change L
```

This state diagram corresponds to the following procedure for reacting to received packets with feature negotiation options. The procedure refers to "P.seqno", "P.ackno", "P.optiontype", and "P.optionlen", which are properties of the packet; "F.GSR" and "F.GSS", which are the variables mentioned in Section 6.6.3; "F.state", which is the feature’s state (STABLE or CHANGING); and "F.value", which is the feature’s value.

```
If F.state == STABLE:
  If P.optiontype == Change R && P.seqno > F.GSR:
    Calculate new value
    Send Confirm L on next packet
    F.GSR := P.seqno
  Otherwise:
    Ignore option

If F.state == CHANGING:
  If P.optiontype == Confirm R && P.ackno >= F.GSS
      && P potentially acknowledges F.GSS:
    If P.optionlen == 3:
      /* empty Confirm R option */
      Retain old value
    Otherwise:
      Check new value
      F.value := new value
    F.state := STABLE
  Otherwise, if P.optiontype == Change R && P.seqno > F.GSR:
    Calculate new value
    Send Confirm L on next packet
    F.GSR := P.seqno
  Otherwise:
    Ignore option
```

7. Sequence Numbers

DCCP uses 24- or 48-bit sequence numbers to arrange packets into sequence, detect losses and network duplicates, and protect against attackers, half-open connections, and the delivery of very old packets. Every packet carries a Sequence Number; most packet types carry an Acknowledgement Number as well.

DCCP sequence numbers are per-packet. Thus, each endpoint increments the DCCP Sequence Number field by one (modulo 2^24 or 2^48) with every packet sent. Even DCCP-Ack and DCCP-Sync packets, and other packets that don't carry user data, increment the Sequence Number. Since DCCP is an unreliable protocol, there are no true retransmissions; but effective retransmissions, such as retransmissions of DCCP-Request packets, also increment the Sequence Number. This lets DCCP implementations detect network duplication, retransmissions, and acknowledgement loss, and is a significant departure from TCP practice.

7.1. Variables

DCCP endpoints maintain a set of sequence number variables for each connection.

ISS The Initial Sequence Number Sent by this endpoint. This equals the Sequence Number of the first DCCP-Request or DCCP-Response sent.

ISR The Initial Sequence Number Received from the other endpoint. This equals the Sequence Number of the first DCCP-Request or DCCP-Response received.

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

GSR The Greatest Sequence Number Received from the other endpoint on an acknowledgeable packet. (Section 7.4 defines "acknowledgeable" packets.)

GAR The Greatest Acknowledgement Number Received from the other endpoint on an acknowledgeable packet.

Some other variables are derived from these primitives.

SWL and SWH
(Sequence Number Window Low and High) The extremes of the validity window for received packets' Sequence Numbers.

AWL and AWH
(Acknowledgement Number Window Low and High) The extremes of the validity window for received packets' Acknowledgement Numbers.

7.2. Initial Sequence Numbers

The endpoints' initial sequence numbers are set by the first DCCP-Request and DCCP-Response packets sent. Initial sequence numbers MUST be chosen to avoid two problems:

- Delivery of old packets, where packets lingering in the network from an old connection are delivered to a new connection with the same addresses and port numbers.
- Sequence number attacks, where an attacker can guess the sequence numbers that a future connection would use [M85].

DCCP implementations may use TCP's strategies for avoiding these problems [RFC 793] [RFC 1948].

To address the first problem, an implementation MUST ensure that the initial sequence number for a given <source address, source port, destination address, destination port> 4-tuple doesn't overlap with recent sequence numbers on connections with the same 4-tuple ("recent" meaning sent within 2 maximum segment lifetimes). If the implementation has state for a recent connection with the same 4-tuple, it can simply pick a good initial sequence number; otherwise, it could tie initial sequence number selection to some clock, such as the 4-microsecond clock used by TCP [RFC 793].

To address the second problem, an implementation MUST provide each 4-tuple with an independent initial sequence number space; then an attacker can't learn anything about anyone else's initial sequence numbers. RFC 1948 achieves this by adding a cryptographic hash, of the 4-tuple and a secret, to any initial sequence number. For the secret, RFC 1948 recommends a combination of some truly-random data [RFC 1750], an administratively-installed passphrase, the endpoint's IP address, and the endpoint's boot time, but truly-random data is sufficient. Care should be taken when changing the secret; such a change alters all initial sequence number spaces, which might make an initial sequence number for some 4-tuple equal a recently sent sequence number for the same 4-tuple. To avoid this problem around such a change, the endpoint might remember dead connection state for each 4-tuple or stay quiet for 2 maximum segment lifetimes.

7.3. Quiet Time

DCCP endpoints, like TCP endpoints, must take care before initiating connections when they boot. In particular, they MUST NOT send packets whose sequence numbers are close to the sequence numbers of packets lingering in the network from before the boot. The simplest way to enforce this rule is for DCCP endpoints to avoid sending any packets until one maximum segment lifetime (2 minutes) after boot. Other enforcement mechanisms include remembering recent sequence numbers across boots, or reserving the upper 8 or so bits of initial sequence numbers for a persistent boot counter that decrements by two each boot (this would require the use of extended sequence numbers).

7.4. Acknowledgement Numbers

DCCP has no cumulative acknowledgement field; cumulative acknowledgements would be meaningless in an unreliable protocol. Therefore, the Acknowledgement Number field has a different meaning in DCCP than in TCP.

A packet is classified as "acknowledgeable" if and only if its options were processed by the receiving DCCP. This means, for example, that all acknowledgeable packets have valid header checksums and sequence numbers. The Acknowledgement Number for most packet types MUST equal GSR, the Greatest Sequence Number Received on an acknowledgeable packet.

Note that "acknowledgeable" refers to option processing, not data processing. Even acknowledgeable packets may have their application data dropped, due to receive buffer overflow or corruption, for instance. Data Dropped options report these data losses when necessary, letting congestion control mechanisms distinguish between network losses and endpoint losses. This issue is discussed further in Sections 11.4 and 11.7.

DCCP-Sync and DCCP-SyncAck packets are a special case to this rule. The Acknowledgement Number on a DCCP-Sync packet corresponds to a received packet, but not necessarily an acknowledgeable packet; in particular, it might correspond to an out-of-sync packet whose options were not processed. The Acknowledgement Number on a DCCP-SyncAck packet always corresponds to an acknowledgeable DCCP-Sync packet; if there was reordering, that Acknowledgement Number might be less than GSR.

7.5. Validity and Synchronization

Any DCCP endpoint might receive packets that are not actually part of the current connection. For instance, the network might deliver an old packet, an attacker might attempt to hijack a connection, or the other endpoint might crash, causing a half-open connection.

DCCP, like TCP, uses sequence number checks to detect these cases Packets whose Sequence and/or Acknowledgement Numbers are out of range are called sequence-invalid, and are not processed normally.

Unlike TCP, DCCP requires a synchronization mechanism to recover from large bursts of loss. One endpoint might send so many packets during a burst of loss that when one of its packets finally got through, the other endpoint would label its Sequence Number as invalid. A handshake involving DCCP-Sync and DCCP-SyncAck packets recovers from this case.

7.5.1. Sequence-Validity Rules

Sequence-validity depends on the received packet's type. This table shows the sequence and acknowledgement number checks applied to each packet; a packet is sequence-valid if it passes both tests, and sequence-invalid if it does not. Many of the checks refer to the sequence and acknowledgement number windows, [SWL, SWH] and [AWL, AWH], defined below in Section 7.5.3.

```
                                                Acknowledgement Number
Packet Type      Sequence Number Check         Check
-----------      ---------------------         ----------------------
DCCP-Request     SWL <= seqno <= SWH (*)       N/A
DCCP-Response    SWL <= seqno <= SWH (*)       AWL <= ackno <= AWH
DCCP-Data        SWL <= seqno <= SWH           N/A
DCCP-Ack         SWL <= seqno <= SWH           AWL <= ackno <= AWH
DCCP-DataAck     SWL <= seqno <= SWH           AWL <= ackno <= AWH
DCCP-CloseReq    SWL <= seqno <= SWH           AWL <= ackno <= AWH
DCCP-Close       SWL <= seqno <= SWH           AWL <= ackno <= AWH
DCCP-Reset       seqno == 0 or seqno > GSR     GAR <= ackno <= AWH
DCCP-Move        seqno >= SWL                  ISS <= ackno <= AWH
DCCP-Sync        seqno >= SWL                  AWL <= ackno <= AWH
DCCP-SyncAck     seqno >= SWL                  AWL <= ackno <= AWH

(*) Check not applied if connection is in LISTEN or REQUEST state.
```

In general, packets are sequence-valid if their Sequence and Acknowledgement Numbers lie within the corresponding valid windows, [SWL, SWH] and [AWL, AWH]. The exceptions to this rule are as follows:

- DCCP-Reset Sequence Numbers may be zero. This is because during the cleanup of a half-open connection, an endpoint might generate a DCCP-Reset in response to a DCCP-Request or DCCP-Data packet with no Acknowledgement Number; the resetting endpoint would then use zero for the Reset's Sequence Number, since it has no valid Sequence Number available.

  DCCP-Reset Acknowledgement Numbers, and non-zero Sequence Numbers, are checked more stringently than those on other packet types, however. This is because DCCP-Reset always ends a connection: no endpoint will send a non-Reset packet on a connection after it has sent a Reset. Thus, a Reset packet whose Sequence Number is less than GSR, or whose Acknowledgement Number is less than GAR, must be sequence-invalid.

- DCCP-Move Sequence and Acknowledgement Numbers are not strongly checked because moves might likely happen after long loss periods, and the mandatory Mobility ID provides good protection against unexpected packets.

- DCCP-Sync and DCCP-SyncAck Sequence Numbers are not strongly checked. These packet types exist specifically to get the endpoints back into sync after bursts of loss; checking their Sequence Numbers would eliminate their usefulness.

These lenient checks all allow continued operation after unusual events, such as endpoint crashes and large bursts of loss. There's no need for leniency when the endpoints are actively sending packets to one another. Therefore, a DCCP endpoint SHOULD implement the following, tighter constraints for active connections. An endpoint considers a connection active if it has received valid packets from the other endpoint within the last several round-trip times, or 1 second, if the RTT is not known.

```
                                                Acknowledgement Number
Packet Type      Sequence Number Check          Check
-----------      ---------------------          ----------------------
DCCP-Reset       GSR <  seqno <= SWH            GAR <= ackno <= AWH
DCCP-Move        SWL <= seqno <= SWH            AWL <= ackno <= AWH
DCCP-Sync        SWL <= seqno <= SWH            AWL <= ackno <= AWH
DCCP-SyncAck     SWL <= seqno <= SWH            AWL <= ackno <= AWH
```

Note that sequence-validity is only one of the validity checks applied to received packets.

7.5.2. Handling Sequence-Invalid Packets

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

When DCCP A receives any other sequence-invalid packet, it MUST reply with a DCCP-Sync packet. This packet MUST acknowledge the packet's Sequence Number (not GSR!). The DCCP-Sync MUST use a new Sequence Number, and thus will increase GSS; GSR will not change, however, since the received packet was sequence-invalid. DCCP A MUST NOT otherwise process sequence-invalid packets. For instance, it MUST NOT process their options.

When the DCCP B endpoint receives the (sequence-valid) DCCP-Sync, it MUST update its GSR variable and reply with a DCCP-SyncAck packet acknowledging the DCCP-Sync (not necessarily GSR!). Upon receiving this DCCP-SyncAck, which will be sequence-valid since it acknowledges the DCCP-Sync, DCCP A will update its GSR variable, and the endpoints will be back in sync. Alternatively, if the connection was half-open (DCCP B is in CLOSED or REQUEST state), DCCP B will send a Reset.

A DCCP endpoint MAY temporarily preserve sequence-invalid packets in case they become valid later. This can reduce the impact of bursts of loss by delivering more packets to the application. In particular, an endpoint MAY preserve a sequence-invalid packet for up to 2 round-trip times (or 1 second, if the RTT is unknown); if, within that time, the relevant sequence windows change so that the packet becomes sequence-valid, the endpoint MAY process the packet again.

To protect itself against denial-of-service attacks (where an attacker sends many sequence-invalid packets, trying to force the receiver to send many DCCP-Syncs), a DCCP implementation MAY rate-limit the DCCP-Syncs sent in response to sequence-invalid packets.

7.5.3. Sequence and Acknowledgement Number Windows

Each DCCP endpoint defines sequence validity windows that are subsets of the Sequence and Acknowledgement Number spaces. These windows correspond to packets the endpoint expects to receive in the next few round-trip times. The Sequence and Acknowledgement Number windows always contain GSR and GSS, respectively; the window widths are controlled by Sequence Window features.

The Sequence Number validity window for packets from DCCP B is [SWL, SWH]. This window always contains GSR, the Greatest Sequence Number Received on a sequence-valid packet from DCCP B. It is W packets wide, where W is the value of the Sequence Window/B feature. One-fourth of the sequence window, rounded down, is placed at and before GSR, with three-fourths after GSR. (This asymmetric placement assumes that bursts of loss are more common in the network than significant reordering.)

```
  invalid  |        valid Sequence Numbers          |  invalid
<---------*|*===========*=======================*|*--------->
      GSR -|GSR + 1 -   GSR                   GSR +|GSR + 1 +
 floor(W/4)|floor(W/4)                   ceil(3W/4)|ceil(3W/4)
           = SWL                              = SWH
```

The Acknowledgement Number validity window for packets from DCCP B is [AWL, AWH]. The high end of the window, AWH, always equals GSS, the Greatest Sequence Number Sent by DCCP A; the window is W' packets wide, where W' is the value of the Sequence Window/A feature.

```
  invalid  |    valid Acknowledgement Numbers     |  invalid
<---------*|*===================================*|*--------->
   GSS - W'|GSS + 1 - W'                       GSS|GSS + 1
           = AWL                              = AWH
```

SWL and AWL are initially adjusted so that they don't go below the initial Sequence Numbers received and sent, respectively:

SWL := max(GSR + 1 - floor(W/4), ISR),

AWL := max(GSS - W' + 1, ISS).

Of course, these adjustments MUST NOT be applied after the relevant sequence numbers wrap.

7.5.4. Sequence Window Feature

The Sequence Window/A feature determines the width of the Sequence Number validity window used by DCCP B, and the width of the Acknowledgement Number validity window used by DCCP A. DCCP A sends a "Change L(Sequence Window, W)" option to notify DCCP B that the Sequence Window/A value is W.

Sequence Window has feature number 3, and is non-negotiable. It takes 3- or 6-byte integer values, like DCCP sequence numbers. Change and Confirm options for Sequence Window are therefore either 6 or 9 bytes long. New connections start with Sequence Window 100 for both endpoints.

A proper Sequence Window/A value should reflect how many packets DCCP A expects to be in flight. Only DCCP A 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. (The next section quantifies this risk.) One good guideline is for each endpoint to set Sequence Window to a small multiple of the maximum number of packets it 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 endpoint can always send updates as the connection progresses.

7.5.5. Sequence Number Attacks

Sequence and Acknowledgement Numbers form DCCP's main line of defense against attackers. An attacker that cannot guess sequence numbers cannot easily manipulate or hijack a DCCP connection, and requirements like careful initial sequence number choice eliminate the most serious attacks.

An attacker might still send many packets with randomly chosen Sequence and Acknowledgement Numbers, however. If one of those probes ends up sequence-valid, it may shut down the connection or otherwise cause problems. The easiest such attacks to execute are:

- Send DCCP-Sync packets with random Sequence and Acknowledgement Numbers. If one of these packets hits the valid acknowledgement number window, the receiver will shift its sequence number window accordingly, getting out of sync with the correct endpoint---perhaps permanently.
- Send DCCP-Reset packets with Sequence Number zero and random Acknowledgement Numbers. If one of these packets hits the valid acknowledgement number window, the connection will be shut down.
- Send DCCP-Data packets with random Sequence Numbers. If one of these packets hits the valid sequence number window, the attack packet's application data may be inserted into the data stream.

The attacker has to guess both Source and Destination Ports for any of these attacks to succeed. Additionally, the connection would have to be inactive for the DCCP-Sync and DCCP-Reset packets to succeed, assuming the victim implemented the more stringent checks for active connections recommended in Section 7.5.1.

To quantify the probability of success, let N be the number of attack packets the attacker is willing to send, W be the relevant sequence window width, and L be the length of sequence numbers (24 or 48). The attacker's best strategy is to space the attack packets evenly over sequence space. Then one of these attacks will succeed with probability P = WN/2^L. For N = 1000, W = 100, and L = 24, this probability is about 0.006. (For reference, the easiest TCP attack---sending a SYN with a random sequence number, which will cause a connection reset if it falls within the window---will succeed with probability 0.002 for N = 1000, W = 8760 [a common default], and L = 32.) Connections with sequence windows much larger than 100 SHOULD use extended sequence numbers to reduce the probability of attack success.

7.5.6. Examples

In the following 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-SyncAck(seq 102, ack 11)   -->   OK
(GSS=102,GSR=11)                                      (GSS=11,GSR=102)
```

In the next 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)        -->    ...
```

7.6. Extended Sequence Numbers

Extended 48-bit sequence numbers increase the rate DCCP connections can achieve without wrapping sequence numbers, and provide additional protection against the sequence number attacks described above. Very-high-rate DCCP connections, and connections with large sequence windows, SHOULD therefore use extended sequence numbers rather than the default 24-bit sequence numbers.

7.6.1. When to Use Extended Sequence Numbers

The sequence-validity mechanism protects against the network delivering old data, but it assumes that the network does not deliver extremely old data. In particular, it assumes that the network must have dropped any packet by the time the connection wraps around and uses its sequence number again. We can easily calculate the maximum connection rate that can be safely achieved given this constraint. Let MSL equal the maximum segment lifetime, P equal the average DCCP packet size in bits, and L equal the length of sequence numbers (24 or 48 bits). Then the maximum safe rate, in bits per second, is R = P*(2^L)/2MSL.

For the default MSL of 2 minutes, 1500-byte DCCP packets, and 24-bit sequence numbers, the safe rate is therefore approximately 800 Mb/s. Of course, 2 minutes is a very large MSL for any networks that could sustain that rate with such small packets. Nevertheless, 48-bit sequence numbers allow much higher rates, up to 14 petabits a second for 1500-byte packets and the default MSL.

The probability of sequence number attack success P = WN/2^L, discussed in Section 7.5.5, may also be relevant when deciding whether to use extended sequence numbers. A fast connection will generally have a relatively high W (sequence window size), increasing the attack success probability for fixed N (number of attack packets); if the probability gets uncomfortably high with L = 24, the connection should use 48-bit sequence numbers instead.

7.6.2. Header Processing

Extended sequence numbers are activated when the header's X bit is set to one (see Section 5.1). This extends the Sequence Number and Acknowledgement Number fields by an additional 24 bits, for a total of 48 bits. The 48-bit numbers are stored in network order, with most significant bit first. All packet types except for DCCP-Data and DCCP-Request will follow this generic header with an extended 48-bit Acknowledgement Number.

Once an endpoint has transitioned to 48-bit sequence numbers (X=1), it MUST send all succeeding packets with 48-bit sequence numbers. Furthermore, once an endpoint has received a sequence-valid packet with 48-bit sequence numbers, it MUST either send all succeeding packets with 48-bit sequence numbers, or reset the connection with Reset Code 7, "Extended Sequence Numbers". (But note that an endpoint may send extended DCCP-Sync packets before transitioning to extended sequence numbers.)

Clients SHOULD decide whether to use extended sequence numbers before sending their DCCP-Requests. However, the Transition bit (T) and Sequence Transition Capable feature support transitioning to extended sequence numbers during an active connection, in case this proves necessary; see below. A client that sends an extended DCCP-Request might receive a DCCP-Reset in response with Reset Code 7, "Extended Sequence Numbers"; the client SHOULD respond by sending another Request using 24-bit sequence numbers.

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, however; see the next section.

7.6.3. 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). The upper 24 bits may be chosen arbitrarily. This applies to Acknowledgement Numbers as well as Sequence Numbers; if DCCP A sends an extended packet containing an Acknowledgement Number before DCCP B sends it a 48-bit Sequence Number, DCCP A can choose any value for the upper 24 bits of the 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.

Switching to 48-bit sequence numbers in the middle of a connection complicates sequence number comparison. Endpoints must compare 48-bit sequence numbers with 24-bit sequence numbers, and compare 48-bit sequence numbers that might have different, arbitrary values in the upper 24 bits, while remaining robust to reordering and to old or malicious packets. The following procedure describes how sequence numbers should be compared during and immediately after a transition.

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, you generally compare them modulo 2^48, except that during a transition, the two values might have arbitrary values in the upper 24 bits.
  - If 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.
  - Otherwise, 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 set, 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.
  - Otherwise, the packet is sequence-invalid.

  Either way, if the packet proves to be sequence-invalid, send an extended DCCP-Sync if 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 sequence-valid packet received from DCCP B had the Transition bit set.
  - 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.
    ```

    The "EL > P" test uses arithmetic comparison, NOT circular comparison. Compare P' with E modulo 2^48.
  - Otherwise, the packet is sequence-invalid.

Either way, if the packet proves to be sequence-invalid, send an extended DCCP-Sync if required, with T set to one.

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.

7.6.4. Sequence Transition Capable Feature

The Sequence Transition Capable feature expresses whether DCCP endpoints are capable of transitioning to extended sequence numbers in the course of an active connection. DCCP A sends a "Change R(Sequence Transition Capable, 1)" option to DCCP B to discover whether B can transition to extended sequence numbers.

Sequence Transition Capable has feature number 4, and is server-priority. It takes one-byte Boolean values. DCCP B MUST allow transitions to extended sequence numbers when Sequence Transition Capable/B is one. It MUST NOT reset the connection with Reset Code 7, "Extended Sequence Numbers", under those circumstances. However, DCCP B MAY allow such transitions even when Sequence Transition Capable/B is zero. Values of two or more are reserved. New connections start with Sequence Transition Capable 0 (that is, not capable) for both endpoints.

7.7. NDP Count and Detecting Application Loss

DCCP's sequence numbers increment by one on every packet, including non-data packets (packets that don't carry application data). This makes DCCP sequence numbers suitable for detecting any network loss, but not for detecting the loss of application data. The NDP Count option reports the length of each burst of non-data packets. This lets the receiving DCCP determine, for every burst of loss, whether or not application data was lost.

```
+--------+--------+-------- ... --------+
|00100101| Length |       NDP Count     |
+--------+--------+-------- ... --------+
 Type=37  Len=3-5
```

If a DCCP endpoint's Send NDP Count feature is one (see below), then that endpoint MUST send an NDP Count option on every packet whose immediate predecessor was a non-data packet. Non-data packets consist of DCCP packet types DCCP-Ack, DCCP-Close, DCCP-CloseReq, DCCP-Reset, DCCP-Move, DCCP-Sync, and DCCP-SyncAck. All other packet types are considered data packets, although not all DCCP-Request and DCCP-Response packets will actually carry application data.

The value stored in NDP Count equals the number of consecutive non-data packets in the run immediately previous to the current packet. Packets with no NDP Count option are considered to have NDP Count zero.

The NDP Count option can carry one to three bytes of data. The smallest option format that can hold the NDP Count SHOULD be used.

7.7.1. Usage Notes

Say that K consecutive sequence numbers are missing in some burst of loss, and the Send NDP Count feature is on. Then some application data was lost within those sequence numbers unless the packet following the hole contains an NDP Count option whose value is greater than or equal to K.

For example, say that the following sequence of non-data packets (Nx) and data packets (Dx) were sent.

```
N0   N1   D2   N3   D4   D5   N6   D7   D8   D9   D10  N11  N12  D13
```

Those packets would have NDP Counts as follows.

```
N0   N1   D2   N3   D4   D5   N6   D7   D8   D9   D10  N11  N12  D13
-    1    2    -    1    -    -    1    -    -    -    -    1    2
```

NDP Count is not useful for applications that include their own sequence numbers with their packet headers.

7.7.2. Send NDP Count Feature

The Send NDP Count feature lets DCCPs negotiate whether they should send NDP Count options on their packets. DCCP A sends a "Change R(Send NDP Count, 1)" option to ask DCCP B to send NDP Count options.

Send NDP Count has feature number 9, and is server-priority. It takes one-byte Boolean values. DCCP B MUST send NDP Count options on its non-data packets (and some of its data packets) when Send NDP Count/B is one, although it MAY send NDP Count options even when Send NDP Count/B is zero. Values of two or more are reserved. New connections start with Send NDP Count 0 for both endpoints.

8. Event Processing

This section describes how DCCP connections move between states, and which packets are sent when. Note that feature negotiation takes place in parallel with the connection-wide state transitions described here.

8.1. Connection Establishment

DCCP connections' initiation phase consists of a three-way handshake: an initial DCCP-Request packet sent by the client, a DCCP-Response sent by the server in reply, and finally an acknowledgement from the client, usually via a DCCP-Ack or DCCP-DataAck packet. The client moves from the REQUEST state to PARTOPEN, and finally to OPEN; the server moves from LISTEN to RESPOND, and finally to OPEN.

```
  Client State                                         Server State
     CLOSED                                               LISTEN
1.   REQUEST   -->       Request        -->
2.             <--       Response       <--   RESPOND
3.   PARTOPEN  -->     Ack, DataAck     -->
4.             <--  Data, Ack, DataAck  <--   OPEN
5.   OPEN      <->  Data, Ack, DataAck  <->   OPEN
```

8.1.1. Client Request

When a client decides to initiate a connection, it enters the REQUEST state, chooses an initial sequence number (Section 7.2), and sends a DCCP-Request packet using that sequence number to the intended server.

DCCP-Request packets will commonly carry feature negotiation options that open negotiations for various connection parameters, such as preferred congestion control IDs for each half-connection. They may also carry application data, but the client should be aware that the server may not accept such data.

A client in the REQUEST state SHOULD send new DCCP-Request packets after some timeout if no response is received. 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 new DCCP-Request MUST increment the Sequence Number by one, and MUST contain the same Service Code and application data as the original DCCP-Request.

A client MAY give up after some number of DCCP-Requests. If so, it SHOULD send a DCCP-Reset packet to the server with Reset Code 2, "Aborted", to clean up state in case one or more of the Requests actually arrived.

The client leaves the REQUEST state for PARTOPEN when it receives a DCCP-Response from the server.

8.1.2. Service Codes

Each DCCP-Request contains a 32-bit Service Code, which identifies the service to which the client application is trying to connect. Service Codes should correspond to application services and protocols. For example, there might be a Service Code for HTTP connections, one for FTP control connections, and one for FTP data connections. Middleboxes, such as firewalls, can use the Service Code to identify the application running on a nonstandard port (assuming the DCCP header has not been encrypted).

Endpoints MUST associate a Service Code with every DCCP socket, both actively and passively opened. The application will generally supply this Service Code. Each active socket MUST have exactly one Service Code, while passive sockets MAY have more than one; this might let multiple applications listen on the same port, differentiated by Service Code. If the DCCP-Request's Service Code doesn't match any of the server's Service Codes for the given port, the server MUST reject the request by sending a DCCP-Reset packet with Reset Code 9, "Bad Service Code". A middlebox MAY also send such a DCCP-Reset in response to packets whose Service Code is considered unsuitable.

Service Codes should be allocated by IANA. We intend for Service Code allocation to be allocated to anyone who asks, first-come first-serve, subject to the following guidelines.

- Service Codes should be allocated one at a time, or in small blocks. A short English description of the intended service is required to obtain a Service Code assignment, but no specification, standards-track or otherwise, is necessary. IANA should maintain an association of Service Codes to the corresponding phrases.
- Users may request specific Service Code values, which 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.
- Service Codes whose bytes each have values in the set {32, 45-57, 65-90} should be reserved for international standard or standards-track specifications, IETF or otherwise. (This set consists of the ASCII digits, uppercase letters, and characters space, '-', '.', and '/'.)
- Service Codes whose high-order byte equals 63 (ASCII '?') should never be allocated. These Service Codes are reserved for private use.
- Service Code 0 should never be allocated. 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.

8.1.3. Server Response

In the second phase of the three-way handshake, the server moves from the LISTEN state to RESPOND, and sends a DCCP-Response message to the client. In this phase, a server will often specify the features 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.

The receiver MAY respond to a DCCP-Request packet with a DCCP-Reset packet to refuse the connection. Relevant Reset Codes for refusing a connection include 8, "Connection Refused", when the DCCP-Request's Destination Port did not correspond to a DCCP port open for listening; 9, "Bad Service Code", when the DCCP-Request's Service Code did not correspond to the service code registered with the Destination Port; and 10, "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 by one, and MUST include the same application data, if any, as the original DCCP-Response.

The responder MUST accept at most one piece of DCCP-Request data per connection. 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).

The Data Dropped and Init Cookie options are particularly useful for DCCP-Response packets (Sections 11.7 and 8.1.4).

The server leaves the RESPOND state for OPEN when it receives a valid DCCP-Ack from the client, completing the three-way handshake.

8.1.4. Init Cookie Option

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

The Init Cookie option lets a DCCP server 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. In the meantime, it need not move from the LISTEN state.

This option is permitted in DCCP-Response, DCCP-Data, DCCP-Ack, DCCP-DataAck, DCCP-Sync, and DCCP-SyncAck packets. 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, meaning the three-way handshake has completed, 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 a tampered Init Cookie option by resetting the connection with Reset Code 11, "Bad Init Cookie".

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.

8.1.5. Handshake Completion

When the client receives a DCCP-Response from the server, it moves from the REQUEST state to PARTOPEN, and completes three-way handshake by sending a DCCP-Ack packet to the server. The PARTOPEN state represents that the client isn't sure whether the server has received any of its DCCP-Acks. The client MUST NOT send DCCP-Data packets while it remains in PARTOPEN. This is because DCCP-Data packets lack Acknowledgement Numbers, so the server can't tell from a DCCP-Data packet whether the client saw its DCCP-Response. Furthermore, if the DCCP-Response included an Init Cookie, that Init Cookie MUST be included on every packet sent in PARTOPEN.

The single DCCP-Ack sent when entering the PARTOPEN state might, of course, be dropped by the network. The client SHOULD ensure that some packet gets through eventually. The preferred mechanism would be a delayed-ack-like 200-millisecond timer, set every time a packet is transmitted in PARTOPEN. If this timer goes off and the client is still in PARTOPEN, the client generates another DCCP-Ack and backs off the timer. If the client remains in PARTOPEN for more than 4MSL, it SHOULD reset the connection with Reset Code 2, "Aborted".

The client leaves the PARTOPEN state for OPEN when it receives a packet other than DCCP-Response or DCCP-Reset from the server.

8.2. Data Transfer

In the central, data transfer phase of the connection, both server and client are in the OPEN state.

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 DCCP-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.

The DCCP-Move, DCCP-Sync, and DCCP-SyncAck packets will also occur in the data transfer phase. DCCP-Move handling is discussed in Section 14, and some cases causing

DCCP-Sync generation are discussed in Section 7.5. One important distinction between DCCP-Sync packets and other packet types is that DCCP-Sync elicits an immediate acknowledgement. On receiving a valid DCCP-Sync packet, a DCCP endpoint MUST immediately generate and send a DCCP-SyncAck in response; and the Acknowledgement Number on that DCCP-SyncAck MUST equal the Sequence Number of the DCCP-Sync.

A particular DCCP implementation might decide to initiate feature negotiation only once the OPEN state was reached, in which case it might not allow data transfer until some time later. Data received during that time SHOULD be rejected and reported using a Data Dropped Drop Block with Drop Code 0.

8.3. Termination

DCCP connection termination uses a handshake consisting of an optional DCCP-CloseReq packet, a DCCP-Close packet, and a DCCP-Reset packet. The server moves from the OPEN state, possibly through the CLOSEREQ state, to CLOSED; the client moves from OPEN through CLOSING to TIMEWAIT, and after 2MSL wait time, to CLOSED.

The sequence DCCP-CloseReq, DCCP-Close, DCCP-Reset is used when the server decides to close the connection, but doesn't want to hold TIMEWAIT state:

```
  Client State                                         Server State
     OPEN                                                  OPEN
1.             <--       CloseReq       <--   CLOSEREQ
2.   CLOSING   -->        Close         -->
3.             <--        Reset         <--   CLOSED
4.   TIMEWAIT
5.   CLOSED
```

A shorter sequence occurs when the client decides to close the connection.

```
  Client State                                         Server State
     OPEN                                                  OPEN
1.   CLOSING   -->        Close         -->
2.             <--        Reset         <--   CLOSED
3.   TIMEWAIT
4.   CLOSED
```

Finally, the server can decide to hold TIMEWAIT state:

```
  Client State                                         Server State
     OPEN                                                  OPEN
1.             <--        Close         <--   CLOSING
2.   CLOSED    -->        Reset         -->
3.                                            TIMEWAIT
4.                                            CLOSED
```

In all cases, the receiver of the DCCP-Reset packet holds TIMEWAIT state for the connection. As in TCP, TIMEWAIT state, where an endpoint quietly preserves a socket for 2MSL (4 minutes) after its connection has closed, ensures that no connection duplicating the current connection's source and destination addresses and ports can start up while old packets might remain in the network.

The termination handshake proceeds as follows. The receiver of a valid DCCP-CloseReq packet MUST respond with a DCCP-Close packet; that receiving endpoint will expect to hold TIMEWAIT state after later receiving a DCCP-Reset. The receiver of a valid DCCP-

Close packet MUST respond with a DCCP-Reset packet, with Reset Code 1, "Closed"; the endpoint that originally sent the DCCP-Close will hold TIMEWAIT state. The endpoint that receives a valid DCCP-Reset packet will hold TIMEWAIT state for the connection.

A DCCP-Reset packet completes every DCCP connection, whether the termination is clean (due to application close; Reset Code 1, "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 TIMEWAIT state for the connection. Processors that must distinguish between clean and unclean termination can examine the Reset Code. DCCP-Reset packets MUST NOT be generated in response to received DCCP-Reset packets. DCCP implementations generally transition to the CLOSED state after sending a DCCP-Reset packet.

Endpoints in the CLOSEREQ and CLOSING states MUST retransmit DCCP-CloseReq and DCCP-Close packets, respectively, until leaving those states. The retransmission timer should initially be set to go off in two RTTs, or 0.4 seconds if the RTT is not known, and should back off to not less than once every 64 RTTs if no relevant response is received.

Only the server can send a DCCP-CloseReq packet or enter the CLOSEREQ state.

8.3.1. Abnormal Termination

DCCP endpoints generate DCCP-Reset packets to terminate connections abnormally; a DCCP-Reset packet may be generated from any state. However, 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 TIMEWAIT states often use Reset Code 3, "No Connection". Resets sent in the REQUEST or RESPOND states often use Reset Code 4, "Packet Error".

8.4. DCCP State Diagram

The most common state transitions discussed above can be summarized in the following state diagram. The diagram is illustrative; the text in Section 8.5 and elsewhere should be considered definitive. For example, there are arcs (not shown) from every state except CLOSED to TIMEWAIT, contingent on the receipt of a valid DCCP-Reset.

```
+---------------------------+    +----------------------------+
|                           v    v                            |
|                        +----------+                         |
|          +-------------+  CLOSED  +-------------+           |
|          |             +----------+  active     |           |
|          | passive                    open      |           |
|          |  open                  snd Request  |           |
|          v                                      v           |
|     +----------+                           +----------+     |
|     |  LISTEN  |                           | REQUEST  |     |
|     +----+-----+                           +----+-----+     |
|          | rcv Request            rcv Response  |           |
|          | snd Response             snd Ack     |           |
|          v                                      v           |
|     +----------+                           +----------+     |
|     | RESPOND  |                           | PARTOPEN |     |
|     +----+-----+                           +----+-----+     |
|          | rcv Ack/DataAck         rcv packet   |           |
|          |                                      |           |
|          |             +----------+             |           |
|          +------------>|   OPEN   |<------------+           |
|                        +--+-+--+--+                         |
|       server active close | |  |   active close             |
|           snd CloseReq    | |  | or rcv CloseReq            |
|                           | |  |    snd Close               |
|                           | |  |                            |
|     +----------+          | |  |           +----------+     |
|     | CLOSEREQ |<---------+ |  +---------->| CLOSING  |     |
|     +----+-----+            |              +----+-----+     |
|          | rcv Close        |                   |           |
|          | snd Reset        |        rcv Reset  |           |
|<---------+                  |                   v           |
|                   rcv Close |              +----+-----+     |
|                   snd Reset |              | TIMEWAIT |     |
|                             |              +----+-----+     |
+-----------------------------+                   |           |
                                                  +-----------+
                                              2MSL timer expires
```

8.5. Pseudocode

This section presents an algorithm describing the processing steps a DCCP endpoint must go through when it receives a packet. A DCCP implementation need not implement the algorithm as it is described here, but any implementation MUST generate observable effects (meaning packets) exactly as indicated by this pseudocode, except where allowed otherwise by another part of this document.

The received packet is written as P, the socket as S. Socket variables:
S.SWL - sequence number window low
S.SWH - sequence number window high
S.AWL - acknowledgement number window low

```
S.AWH - acknowledgement number window high
S.ISS - initial sequence number sent
S.ISR - initial sequence number received
S.OSR - first OPEN sequence number received
S.GSS - greatest sequence number sent
S.GSR - greatest valid sequence number received
S.GAR - greatest acknowledgement number received; initialized to S.ISS
"Send packet" actions always use, and increment, S.GSS.

First, check the header basics;
  If the header checksum is incorrect, drop packet and return.
  If the packet type is not understood, drop packet and return.
  If Data Offset is too small for packet type, or too large for packet,
  drop packet and return.

Second, process DCCP-Move;
  If P.type == Move,
    Look up the Mobility ID in table; get socket.
    If socket exists && P.seqno >= S.SWL && P.ackno <= S.AWH
        && P.ackno >= S.ISS && S.state >= PARTOPEN && S.state < TIMEWAIT,
      Process options
      Set socket to point at new address/ports
      Add reference to new address/ports
      Set timer to remove old address/ports after 2MSL
      Choose new Mobility ID, add to table
      Send DCCP-Sync[Change L[Mobility ID, new ID]]
      Update S.GSR, S.SWL, S.SWH
      Drop packet and return
    Otherwise,
      Drop packet and return

Third, check ports and process TIMEWAIT state;
  Look up flow ID; get socket.
  If no socket, or S.state == TIMEWAIT,
    Generate Reset(No Connection) unless P.type == Reset
    Drop packet and return

Fourth, process LISTEN state;
  If S.state == LISTEN,
    If P.type == Request,
      /* Init Cookie processing would go here */
      Set S := new socket for this port pair
      S.state = RESPOND
      Choose S.ISS (initial seqno)
      Set S.ISR, S.GSR, S.SWL, S.SWH from packet
      Continue (with S.state == RESPOND)
    Otherwise,
      Generate Reset(No Connection) unless P.type == Reset
      Drop packet and return
```

```
Fifth, process Reset;
  If P.type == Reset,
    If S.GAR <= P.ackno <= S.AWH
        && (P.seqno == 0 || P.seqno > S.GSR || S.state == REQUEST),
      Tear down connection
      S.state := TIMEWAIT
      Set TIMEWAIT timer
      Drop packet and return
    Otherwise (sequence numbers out of whack),
      Drop packet and return

Sixth, process REQUEST state;
  If S.state == REQUEST,
    If P.type == Response && S.AWL <= P.ackno <= S.AWH,
      Set S.GSR, S.ISR, S.SWL, S.SWH
    Otherwise,
      Generate Reset(Packet Error)
      Drop packet and return

Seventh, process Sync sequence numbers;
  If P.type == Sync || P.type == SyncAck,
    If S.AWL <= P.ackno <= S.AWH and P.seqno >= S.SWL,
      Update S.GSR, S.SWL, S.SWH
    Otherwise,
      Drop packet and return

Eighth, check sequence numbers;
  If S.SWL <= P.seqno <= S.SWH
      && (P.ackno does not exist || S.AWL <= P.ackno <= S.AWH),
    Update S.GSR, S.GAR, S.SWL, S.SWH
  Otherwise,
    Send Sync packet acknowledging P.seqno
    Drop packet and return

Ninth, check packet type;
  If (S.is_server && P.type == CloseReq)
     || (S.is_server && P.type == Response)
     || (S.is_client && P.type == Request)
     || (S.state >= OPEN && P.type == Request && P.seqno >= S.OSR)
     || (S.state >= OPEN && P.type == Response && P.seqno >= S.OSR)
     || (S.state == RESPOND && P.type == Data),
    Send Sync packet acknowledging P.seqno
    Drop packet and return

Tenth, process options;
  /* may involve resetting connection, etc. */
  Mark packet as "received" for acknowledgement purposes
  On processing Confirm R(Mobility ID),
    Check that the confirmed Mobility ID is correct
    If a DCCP-Move was recently processed,
```

```
        Remove any old Mobility ID from table

  Eleventh, process RESPOND state;
    If S.state == RESPOND,
      If P.type == Request,
        Send Response
      Otherwise,
        S.OSR := P.seqno
        S.state := OPEN

  Twelfth, process REQUEST state;
    If S.state == REQUEST,
      S.state := PARTOPEN
      /* Do not send Data packets in PARTOPEN; furthermore, include Init
        Cookie on every packet */
      Set PARTOPEN timer

  Thirteenth, process PARTOPEN state;
    If S.state == PARTOPEN,
      If P.type == Response,
        Send Ack
      Otherwise,
        S.OSR := P.seqno
        S.state := OPEN

  Fourteenth, process CloseReq;
    If P.type == CloseReq && S.state < CLOSEREQ,
      Generate Close
      S.state := CLOSING
      Set CLOSING timer

  Fifteenth, process Close;
    If P.type == Close,
      Generate Reset(Closed)
      Tear down connection
      Drop packet and return

  Sixteenth, process Sync;
    If P.type == Sync,
      Generate SyncAck

  Seventeenth, process data.
    Do not deliver data from more than one Request or Response
```

9. Checksums

DCCP uses a header checksum to protect its header against corruption. Generally, this checksum covers any application data as well. However, DCCP applications can request that the header checksum cover only part of the application data, or perhaps no application data at all. Link layers may then reduce their protection on unprotected parts of DCCP packets. For some noisy links, and applications that can tolerate corruption, this can greatly improve delivery rates and perceived performance.

If checksum coverage is complete, packets with corrupt application data must be treated as network losses, thus incurring a loss response from the sender's congestion control mechanism. Such a heavy-duty response may unfairly penalize connections on links with high background corruption. It is to the application's benefit to report corruption losses differently from network losses. Therefore, even applications that demand correct data can make use of reduced checksum coverage, by including a Data Checksum option. Data Checksum holds a strong checksum of the application data. The combination of reduced checksum coverage and Data Checksum can detect application data corruption, but report it as corruption, not congestion, via Data Dropped options (see Section 11.7).

Reduced checksum coverage introduces some security considerations; see Section 19.2. See Appendix B.1 for further motivation and discussion. DCCP's implementation of reduced checksum coverage was inspired by UDP-Lite [UDP-LITE].

9.1. Header Checksum Field

DCCP uses the TCP/IP checksum algorithm. The Checksum field in the DCCP generic header (see Section 5.1) 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 application data. 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.

9.2. Header Checksum Coverage Field

The Checksum Coverage field in the DCCP generic header (see Section 5.1) specifies what parts of the packet are covered by the Checksum field, as follows:

CsCov = 0 The Checksum field covers the DCCP header, DCCP options, network-layer pseudoheader, and all application data in the packet, possibly padded on the right with zeros to an even number of bytes.

CsCov = 1-15 The Checksum field covers the DCCP header, DCCP options, network-layer pseudoheader, and the initial (CsCov-1)*4 bytes of the packet's application data.

Thus, if CsCov is 1, none of the application data is protected by the header checksum. The value (CsCov-1)*4 MUST be less than or equal to the length of the application data. 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 application data. In fact, DCCP cannot even detect corruption in areas not covered by the header checksum, unless the Data Checksum option is used. Applications should not make any assumptions about the correctness of received data not covered by the checksum, and should if necessary introduce their own 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 application data should be accepted. (Note that, for short packets, application data might be fully covered by a nonzero Checksum Coverage value.) 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].

9.3. Data Checksum Option

The Data Checksum option holds a 32-bit CRC-32c cyclic redundancy-check code of a DCCP packet's application data.

```
+--------+--------+--------+--------+--------+--------+
|00101100|00000110|              CRC-32c              |
+--------+--------+--------+--------+--------+--------+
 Type=44  Length=6
```

Data Checksum is intended for packets containing application data, such as DCCP-Request, DCCP-Response, DCCP-Data, and DCCP-DataAck, but it may be included on any packet. The sending DCCP computes the CRC of the bytes comprising the application data and stores it in the option data. The CRC-32c algorithm used for Data Checksum is the same as that used for SCTP [RFC 3309]; note that the CRC-32c of zero bytes of data equals zero. The DCCP header checksum will cover the Data Checksum option, so the data checksum must be computed before the header checksum.

The receiving DCCP SHOULD compute the received application data's CRC-32c using the same algorithm as the sender, and compare the result and the Data Checksum value. If the values differ, the packet's application data MUST be dropped, and reported using a Data Dropped option as dropped due to corruption (Drop Code 3). However, 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 similar options.

9.3.1. Check Data Checksum Feature

The Check Data Checksum feature lets a sending DCCP determine whether or not its partner can check Data Checksum options. DCCP A sends a Mandatory "Change R(Check Data Checksum, 1)" option to DCCP B to require B to check Data Checksum options (the connection will be reset if DCCP B cannot).

Check Data Checksum has feature number 10, and is server-priority. It takes one-byte Boolean values. DCCP B MUST check any received Data Checksum options when Check Data Checksum/B is one, although it MAY check them even when Check Data Checksum/B is zero. Values of two or more are reserved. New connections start with Check Data Checksum 0 for both endpoints.

9.3.2. Usage Notes

Internet links must normally apply strong integrity checks to the packets they transmit [UDP-LITE] [LINK BCP]. Data Checksum is redundant for DCCP packets whose integrity is checked by every link they traverse. This is the default case when the DCCP header's Checksum Coverage value equals zero (full coverage). However, the DCCP Checksum Coverage value might not be zero. By setting partial Checksum Coverage, the application indicates that it can tolerate corruption in the unprotected part of the application data. Recognizing this, link layers may reduce the strength of their error detection and/or correction when transmitting this unprotected part, which can significantly increase the probability of the endpoint receiving corrupt data. Data Checksum lets the receiver detect any ensuing corruption.

10. 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 CCID/A value equals the CCID in use for the A-to-B half-connection. DCCP B sends a "Change R(CCID, K)" option to ask DCCP A to use CCID K for its data packets.

CCID is a server-priority feature, so CCID negotiation options can list multiple 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
```

New connections start with CCID 2 for both endpoints. If this is unacceptable for a DCCP endpoint, that endpoint MUST send Mandatory Change(CCID) options on its first packets.

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, such as an implementation in a general-purpose operating system kernel, SHOULD implement at least CCIDs 1 and 2. The intent is to make these CCIDs broadly available for interoperability, although particular applications might disallow their use.

10.1. Unspecified Sender-Based Congestion Control

CCID 1 denotes an unspecified sender-based congestion control mechanism. This provides a limited, controlled form of interoperability for new IETF-approved CCIDs: with CCID 1, an HC-Sender can use a new sender-based congestion control mechanism whose details the HC-Receiver does not understand.

Some congestion control mechanisms require only generic behavior from the receiver. For example, CCID 2, TCP-like Congestion Control, requires that the receiver (1) send Ack Vectors and (2) respond to Ack Ratio. Both of these requirements use generic mechanisms described in this document. Thus, a CCID 2 HC-Receiver doesn't really need to understand the details of CCID 2.

CCID 1 uses this insight to support forward compatibility for sender-based congestion control mechanisms. An HC-Sender proposes CCID 1 as a proxy for a sender-based mechanism whose details the HC-Receiver doesn't need to understand. The HC-Receiver can then agree to CCID 1, and provide generic acknowledgement feedback as requested by other features (such as Send Ack Vector). Individual CCID profile documents say whether or not they can masquerade as CCID 1.

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 proxy for CCID 98. Now, say DCCP A would like to use CCID 98 for its data packets. It should therefore send a "Change L(CCID, 98 1)" option to open a CCID negotiation. 98 comes first, since that is the preferred CCID; 1 comes next, as a potential proxy for 98. If DCCP B understands CCID 98, it will respond with "Confirm R(CCID, 98, ...)" and all is well. But if it does not understand CCID 98, it may respond with "Confirm R(CCID, 1, ...)", still allowing DCCP A to use CCID 98. DCCP A will separately negotiate Send Ack Vector, and thus DCCP B will provide the feedback DCCP A requires, namely Ack Vector, without needing to understand the operation of CCID 98.

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.

Since CCID 1 should be used only as a proxy for other, defined CCIDs, an HC-Sender MUST NOT report a preference list consisting only of CCID 1, and the option "Change L(CCID, 1)" is illegal. Receiving such an option SHOULD result in connection reset with Reset Code 5, "Option Error". An HC-Receiver MAY suggest CCID 1 exclusively: the option "Change R(CCID, 1)" is not illegal.

If CCID 1 is the result of a CCID feature negotiation, the HC-Sender determines which CCID to actually use by picking the earliest CCID in its preference list that can masquerade as CCID 1. The HC-Sender MUST pick a CCID that appeared explicitly in its preference list.

Many DCCP APIs will allow applications to suggest preferred CCIDs for sending and receiving data. Such APIs might let applications allow or prevent the use of CCID 1 for receiving, but they should not let applications suggest the use of CCID 1 for sending. The code implementing a particular CCID should add CCID 1 to the HC-Sender's CCID preference list when appropriate, unless the application disagrees. The default for both sender and receiver should be to allow CCID 1 when possible.

CCID 1 places no restrictions on how often the HC-Receiver may send DCCP-Ack packets. A careful implementation SHOULD implement a liberal rate limit on DCCP-Acks to prevent ack storms.

10.2. 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].

10.3. 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].

10.4. CCID-Specific Options, Features, and Reset Codes

Half of the option types, feature numbers, and Reset Codes are reserved for CCID-specific use. CCIDs may often need new options, for communicating acknowledgement or rate information, for example; reserved option spaces let CCIDs 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 Codes 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 Codes 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 Codes 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:

Packet	Option	Relevant Half-conn.	Relevant 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 lists CCID X first. 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, except that Mandatory CCID-specific options and features MUST induce a DCCP-Reset with Reset Code 6, "Mandatory Error".

11. 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 ack 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.

11.1. 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 B can free old Ack Vector state. For instance, 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.

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.

11.2. 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 allowed maximum packet size. 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.

11.3. 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 7, and is non-negotiable. It takes two-byte integer values. The Ack Ratio/A feature is the rough ratio of data packets sent by DCCP A to acknowledgement packets sent back by DCCP B. For example, if Ack Ratio/A is four, then DCCP B will send at least one acknowledgement packet for every four data packets sent by DCCP A. DCCP A sends a "Change L(Ack Ratio)" option to notify DCCP B of its ack ratio. New connections start with Ack Ratio 2 for both endpoints.

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.

11.4. 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 12.2 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
+-+-+-+-+-+-+-+-+
|Sta| Run Length|
+-+-+-+-+-+-+-+-+
```

Sta[te] 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.

Run Length, the least significant six bits of each byte, specifies how many consecutive packets have the given State. Run Length zero says the corresponding State applies to one packet only; Run Length 63 says it applies to 64 consecutive packets. Run lengths of 65 or more must be encoded in multiple bytes.

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).

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 begins where the first 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 application data 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 Reset Code 5, "Option Error". Packets that haven't been included in 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.

11.4.1. 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. (Information about a sequence number is generally repeated in every ack until the HC-Sender acknowledges an ack. In this case, 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).

11.4.2. 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.

11.5. Send Ack Vector Feature

The Send 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. Send Ack Vector is mandatory for some CCIDs, and optional for others.

Send Ack Vector has feature number 8, and is server-priority. It takes one-byte Boolean values. DCCP A MUST send Ack Vector options on its acknowledgements when Send

Ack Vector/A has value one, although it MAY send Ack Vector options even when Send Ack Vector/A is zero. Values of two or more are reserved. New connections start with Send Ack Vector 0 for both endpoints. DCCP B sends a "Change R(Send Ack Vector, 1)" option to DCCP A to ask A to send Ack Vector options as part of its acknowledgement traffic.

11.6. 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. 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.

11.7. 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 application data 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 data 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 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 Reset Code 5, "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).

11.7.1. 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.

11.7.2. 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.

12. Explicit Congestion Notification

The DCCP protocol is fully ECN-aware [RFC 3168]. 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.

12.1. 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, and is server-priority. It takes one-byte Boolean values. DCCP A MUST be able to read ECN bits from received frames' IP headers when ECN Capable/A is one. (This is independent of whether it can set ECN bits on sent frames.) DCCP A thus sends a "Change L(ECN Capable, 0)" option to DCCP B to inform it that A cannot read ECN bits. New connections start with ECN Capable 1 (that is, ECN capable) for both endpoints. Values of two or more are reserved.

If a DCCP is not ECN capable, it MUST send Mandatory "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.

12.2. ECN Nonces

Congestion avoidance will not occur, and the receiver will sometimes get its data faster, if the sender isn't told about 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.)

Given an A-to-B half-connection, 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 with Reset Code 12, "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.)

12.3. 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 Reset Code 12, "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.

13. Timing Options

The Timestamp, Timestamp Echo, and Elapsed Time options help DCCP endpoints explicitly measure round-trip times.

13.1. 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.

13.2. 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.

```
+--------+--------+--------+--------+
|00101011|00000100|   Elapsed Time  |
+--------+--------+--------+--------+
 Type=43    Len=4

+--------+--------+--------+--------+--------+--------+
|00101011|00000110|            Elapsed Time           |
+--------+--------+--------+--------+--------+--------+
 Type=43    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 Reset Code 12, "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, it allows most reasonable elapsed times to fit into two bytes of data.

13.3. 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 value, similar to that in the Elapsed Time option, 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.

14. 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. 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, there's 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]. DCCP mobility may not be useful in the context of IPv6, with its mandatory support for Mobile IP.

14.1. 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. 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 has feature number 5, and is server-priority. It takes one-byte Boolean values. DCCP A agrees in principle to accept DCCP-Move packets from DCCP B when Mobility Capable/A is one. DCCP A MUST reject any DCCP-Move packet for a connection whose Mobility Capable/A feature is zero, although it MAY reject a valid DCCP-Move packet even when Mobility Capable/A is one. Values of two or more are reserved. New connections start with Mobility Capable 0 (that is, mobility is not allowed) for both endpoints.

14.2. Mobility ID Feature

A DCCP uses the Mobility ID feature to inform its partner of a 128-bit number that will act as identification, should the partner change its address and/or port during the course of the connection. DCCP A sends a "Change L(Mobility ID, N)" option to notify DCCP B of the ID it has chosen for B's use.

Mobility ID has feature number 6, and is non-negotiable. Its values are sixteen-byte integers. The Mobility ID/A feature equals the identifier that DCCP B should use on DCCP-Move packets sent to A. DCCP A chooses Mobility ID/A to uniquely identify the connection among all connections that terminate at A. For security, DCCP A MUST choose Mobility ID/A randomly. Furthermore, it MUST reassign Mobility ID/A after each successful move by DCCP B, and it MAY reassign Mobility ID/A more frequently. New connections start with Mobility ID 0 for both endpoints. However, Mobility IDs of zero MUST NOT be accepted on DCCP-Move packets; an endpoint cannot successfully move until the relevant Mobility ID has been set to a nonzero value.

14.3. Mobile Host Processing

When DCCP A changes its address and/or port, it MUST signal this by sending DCCP B a DCCP-Move packet. The Mobility ID in the DCCP-Move packet uniquely identifies the connection; DCCP B will read the new address and port off the DCCP-Move's network and DCCP headers. Eventually, DCCP A will receive a DCCP-Sync sent to its new address that negotiates a new Mobility ID/B feature. This confirms the move. DCCP A SHOULD retransmit the DCCP-Move packet until it receives a DCCP-Sync confirmation. The retransmission strategy SHOULD be similar to that for retransmitting DCCP-Requests (Section 8.1.1); for instance, a first timeout on the order of a second, with an exponential backoff timer.

DCCP A MUST reset its congestion control state after sending a DCCP-Move, since nothing is known about conditions on the new path. Essentially, DCCP A must "slow start" up to its new fair rate, as appropriate for its congestion control mechanism. Section 14.5 discusses this further.

DCCP A SHOULD NOT send non-DCCP-Move packets to DCCP B until the move is confirmed. If it did so, and the DCCP-Move packet was lost or reordered, then DCCP B would react by sending DCCP-Resets with Reset Code 3, "No Connection". DCCP A might implement special handling for such resets to avoid any post-move quiet period, but this is NOT RECOMMENDED.

DCCP B MAY refuse to accept a move, perhaps because of address policy. In this case, DCCP A will receive a DCCP-Reset with Reset Code 13, "Move Refused", rather than a confirming DCCP-Sync. DCCP A MAY react by tearing down the connection, or by trying another DCCP-Move---for instance, back to the old address, if possible.

DCCP endpoints SHOULD NOT use an old address-port pair after sending a DCCP-Move. If it becomes necessary to switch back to the old address-port pair, the endpoint MUST do so explicitly using another DCCP-Move.

DCCP-Move packets SHOULD NOT be sent until the connection is established; it is illegal to send a DCCP-Move in REQUEST or RESPOND state. If an endpoint moves during connection establishment, it SHOULD abandon the old connection and initiate a new one. No connection exists to move until the three-way handshake has completed.

14.4. Stationary Host Processing

The stationary endpoint, DCCP B, uses DCCP-Move packets' destination address, destination port, and Mobility ID fields to look up the relevant connection. This differs from all other packet types, which use the source address/source port/destination address/destination port 4-tuple.

DCCP B MUST ignore DCCP-Moves whose Mobility ID is zero, or whose Mobility ID does not correspond to any active connection. It also MUST ignore DCCP-Moves sent to sockets in CLOSED, LISTEN, REQUEST, RESPOND, or TIMEWAIT state, and it MUST ignore DCCP-Moves with invalid Sequence or Acknowledgement Numbers (see Section 7.5). DCCP B MUST NOT respond to invalid DCCP-Moves with DCCP-Reset or DCCP-Sync packets, since any active response would leak information about the connection 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.

On receiving a valid DCCP-Move, DCCP B decides whether to accept or refuse the move request. To accept the request, it performs several actions:

- It changes the connection to use the new address and port.
- It sets a timer to remove the old address and port after 2MSL. This delay allows the receipt of any delayed packets from the old address and port, and essentially represents TIMEWAIT state for the old connection.
- It chooses a new Mobility ID for the connection, which temporarily coexists with the old Mobility ID.
- It generates and sends a confirmation DCCP-Sync packet, which includes a "Change L(Mobility ID)" option for the new Mobility ID.

If the DCCP-Sync is lost, then DCCP A will send another DCCP-Move packet with the old Mobility ID. DCCP B MUST send another DCCP-Sync packet in this situation, but SHOULD NOT choose yet another new Mobility ID.

The move's three-way handshake completes once DCCP B receives a DCCP-SyncAck from DCCP A that confirms the new Mobility ID option. At that point, DCCP B MUST remove the old Mobility ID.

DCCP B MAY refuse a valid DCCP-Move request for any reason; for instance, the new address space might be considered unsuitable. To refuse a valid DCCP-Move, DCCP B sends a DCCP-Reset packet to the new address and port pair with Reset Code 13, "Move Refused". It need take no other action; for example, it MAY tear down the connection, or not. If DCCP B plans to refuse every DCCP-Move request, it MUST negotiate a zero value for the Mobility Capable/A feature.

DCCP B MUST ignore any data following the header in a DCCP-Move packet.

14.5. 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' PMTUs SHOULD be reinitialized, and PMTU discovery performed again, following an address change. See Section 15.

During the transition period between addresses, the endpoints might receive congestion feedback from both before the move and after the move. Congestion and loss events on packets sent before the move SHOULD NOT affect the new connection's congestion control state.

14.6. Security

The DCCP mobility mechanism, like DCCP in general, does not provide cryptographic security guarantees. Nevertheless, mobile hosts must use valid Mobility IDs, providing protection against some classes of attackers: An attacker cannot move a DCCP connection to a new address unless it knows a valid Mobility ID. This generally means that an attacker must have snooped on every packet in the connection to get a reasonable probability of success, assuming that the Mobility ID was chosen well (that is, randomly).

An attacker could choose a server running many mobility-capable connections, and simply guess random Mobility IDs until one hit. Let N equal the number of mobility-capable connections at the server, X equal the number of attack attempts, and D equal the number of possible Mobility IDs, namely 2^128. Then the probability of at least one attack succeeding is

```
                (D - N) choose X           (D-N)! (D-X)!
 P  =  1  -  ----------------  =  1  -  ------------- .
                  D choose X                D! (D-N-X)!
```

For N = 10^6 and X = 10^9, the attack success probability is less than 10^-23.

Section 19 further describes DCCP security considerations.

15. 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 on IPv4 networks. 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 classical PMTU discovery, the DCCP sender sets the IP Don't Fragment (DF) bit. However, it is undesirable 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 refuse to send the packet and return an appropriate error to the application. The application should then use the API to query the new value of MPS. The kernel might have some packets buffered for transmission that are smaller than the old MPS, but larger than the new MPS. It 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 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.

A DCCP implementation SHOULD ensure, as far as possible, that ICMP Datagram Too Big messages were actually generated by routers, so that attackers cannot drive the PMTU down to a falsely small value. The simplest way to do this is to verify that the Sequence Number on the ICMP error's encapsulated header corresponds to a Sequence Number that the implementation recently sent. (Routers are not required to return more than 64 bits of the DCCP header [RFC 792], but most modern routers will return far more, including the Sequence Number.) ICMP Datagram Too Big messages with incorrect or missing Sequence Numbers may be ignored, or the DCCP implementation may lower the PMTU only temporarily in response. If more than three odd Datagram Too Big messages are received and the other DCCP endpoint reports commensurate loss, however, the DCCP implementation SHOULD assume the presence of a confused router, and either obey the ICMP messages' PMTU or (on IPv4 networks) switch to allowing fragmentation.

DCCP also allows upward probing of the PMTU [PMTUD], where the DCCP endpoint begins by sending small packets with DF set, then gradually increases the packet size until a packet is lost. This mechanism does not require any ICMP error processing. DCCP-Sync packets are the best choice for upward probing, since DCCP-Sync probes do not risk application data loss. The DCCP implementation inserts arbitrary data into the DCCP-Sync application area, padding the packet to the right length; and since every valid DCCP-Sync generates an immediate DCCP-SyncAck in response, the endpoint will have a pretty good idea of when a probe is lost.

16. Forward Compatibility

Future versions of DCCP may add new options and features. A few simple guidelines will let extended DCCPs interoperate with normal DCCPs.

- DCCP processors MUST NOT act punitively towards options and features they do not understand. For example, DCCP processors MUST NOT reset the connection if some field marked Reserved in this specification is non-zero; if some unknown option is present; or if some feature negotiation option mentions an unknown feature. Instead, DCCP processors MUST ignore these events. The Mandatory option is the single exception: if Mandatory precedes some unknown option or feature, the connection MUST be reset.
- DCCP processors MUST anticipate the possibility of unknown feature values, which might occur as part of a negotiation for a known feature. For server-priority features, unknown values are handled as a matter of course: since the non-extended DCCP's priority list will not contain unknown values, the result of the negotiation cannot be an unknown value. A DCCP SHOULD reset the connection if it is assigned an unacceptable value for some non-negotiable feature.
- Each DCCP extension SHOULD be controlled by some feature. The default value of this feature should correspond to "extension not available". If an extended DCCP wants to use the extension, it SHOULD attempt to change the feature's value using a Change L or Change R option. Any non-extended DCCP will ignore the option, thus leaving the feature value at its default, "extension not available".

Section 20 lists DCCP assigned numbers reserved for experimental and testing purposes.

17. 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 Reset Code 9, "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 acknowledgement options, such as Ack Vector, and CCID-specific options, such as TFRC's Loss Intervals, at minimum. 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". Furthermore, if a middlebox completely changed sequence numbers, the DCCP-Move mobility mechanism might stop working. 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.

The contents of this section should not be interpreted as a wholesale endorsement of stateful middleboxes.

18. Relations to Other Specifications

18.1. 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). Therefore, it is important to examine 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. 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 makes the other redundant.

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.

18.2. Multiplexing Issues

Since DCCP doesn't provide reliable, ordered delivery, multiple application sub-flows may be multiplexed over a single DCCP connection with no inherent performance penalty. Thus, there is no need for DCCP to provide built-in, SCTP-style support for multiple sub-flows.

Some applications might want to share congestion control state among multiple DCCP flows that share the same source and destination addresses. This functionality could be provided by the Congestion Manager [RFC 3124], a generic multiplexing facility. However, the CM would not fully support DCCP without change; it does not gracefully handle multiple congestion control mechanisms, for example.

19. Security Considerations

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

Nevertheless, DCCP is intended to protect against some classes of attackers: Attackers cannot hijack a DCCP connection (close the connection unexpectedly, or cause attacker data to be accepted by an endpoint as if it came from the sender) unless they can guess valid sequence numbers. Thus, as long as endpoints choose initial sequence numbers well, a DCCP attacker must snoop on data packets to get any reasonable probability of success. Sequence number validity checks provide this guarantee. Section 7.5.5 describes sequence number security further.

This security property only holds assuming that DCCP's random numbers are chosen according to the guidelines in [RFC 1750].

DCCP provides no protection against attackers that can snoop on data packets.

19.1. Security Considerations for Mobility

Mobility slightly changes DCCP's security properties 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, since it must guess a valid Mobility ID. Section 14.6 quantifies the probability of successful attack; with DCCP's 128-bit Mobility IDs, that probability is quite low.

19.2. Security Considerations for Partial Checksums

The partial checksum facility has a 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 application data.

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.

20. 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 Codes (Section 5.6).
- 8-bit DCCP Option Types (Section 5.9). 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). 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 10).
- Ack Vector States (Section 11.4). Only State 2 remains unallocated.
- Data Dropped Drop Codes 4 through 6 (Section 11.7).

IANA should also provide a registry for 32-bit Service Codes. Registering a Service Code should not require a standards-track specification. Our liberal proposed registration rules for Service Codes are presented in detail in Section 8.1.2.

Finally, DCCP requires a Protocol Number to be added to the registry of Assigned Internet Protocol Numbers. Protocol Number 33 has informally been made available for experimental DCCP use, but this number may change in future.

The following DCCP assigned numbers should be reserved specifically for experimental and testing use [RFC 3692]: packet type 15, option number 31, option numbers 120 through 126, feature numbers 120 through 126, Reset Codes 248 through 254, and CCID 254.

21. Thanks

Thanks to Jitendra Padhye for his help with early versions of this specification.

Thanks to Junwen Lai and Arun Venkataramani, who, as interns at ICIR, built a prototype DCCP implementation. In particular, Junwen Lai recommended that the old feature negotiation mechanism be scrapped and helped design the current mechanism, and Arun Venkataramani's feedback improved Appendix A.

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 Data 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

           <=== buf_head and buf_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 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 buf_head to ack_ackno, 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). It changes as information about old acknowledgements is removed (so ack_ptr and buf_head diverge), and 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 11.4.1 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_ptr will equal buf_head; 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 ack_nonce for every 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 should check the ECN Nonce for correctness. (As described in Section 11.4.1, 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 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

This section attempts 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 application data. If the application cannot tolerate any data 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 application data, it may do so by including the Data Checksum option. This would allow DCCP to discard corrupted application data, 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 application data. 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 application data 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 1750] D. Eastlake, S. Crocker, and J. Schiller. Randomness Recommendations for Security. RFC 1750.

[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.

[RFC 3309] J. Stone, R. Stewart, and D. Otis. Stream Control Transmission Protocol (SCTP) Checksum Change. RFC 3309.

[RFC 3692] T. Narten. Assigning Experimental and Testing Numbers Considered Useful. RFC 3692.

[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.

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-ccid2-05.txt, work in progress, February 2004.

[CCID 3 PROFILE] S. Floyd, E. Kohler, and J. Padhye. Profile for DCCP Congestion Control ID 3: TFRC Congestion Control. draft-ietf-dccp-ccid3-05.txt, work in progress, February 2004.

[LINK BCP] Phil Karn, editor. Advice for Internet Subnetwork Designers. draft-ietf-pilc-link-design-13.txt, work in progress, February 2003.

[M85] Robert T. Morris. A Weakness in the 4.2BSD Unix TCP/IP Software. Computer Science Technical Report 117, AT&T Bell Laboratories, Murray Hill, NJ, February 1985.

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

[RFC 792] J. Postel, editor. Internet Control Message Protocol. RFC 792.

[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 3517] E. Blanton, M. Allman, K. Fall, and L. Wang. A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP. RFC 3517.

[RFC 3540] N. Spring, D. Wetherall, and D. Ely. Robust Explicit Congestion Notification (ECN) Signaling with Nonces. RFC 3540.

[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.

Authors' Addresses

Eddie Kohler <kohler@cs.ucla.edu>
4531C Boelter Hall
UCLA Computer Science Department
Los Angeles, CA 90095
USA

Mark Handley <M.Handley@cs.ucl.ac.uk>
Department of Computer Science
University College London
Gower Street
London WC1E 6BT
UK

Sally Floyd <floyd@icir.org>
ICSI Center for Internet Research
1947 Center Street, Suite 600
Berkeley, CA 94704
USA

Intellectual Property Notice

Full Copyright Statement