Internet Engineering Task Force
INTERNET-DRAFT
draft-kohler-dcp-01.ps

Eddie Kohler
Mark Handley
Sally Floyd
Jitendra Padhye
ACIRI

21 November 2001
Expires: May 2002

Datagram Control Protocol (DCP)

Status of this Document

Abstract

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

Table of Contents

1. Introduction

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

- An unreliable flow of datagrams, with acknowledgements.
- A reliable handshake for connection setup and teardown.
- Reliable negotiation of options, including negotiation of a suitable congestion control mechanism.
- Mechanisms allowing a server to avoid holding any state for unacknowledged connection attempts or already-finished connections.
- An optional mechanism that allows the sender to know, with high reliability, which packets reached the receiver.
- Congestion control incorporating Explicit Congestion Notification (ECN) and the ECN Nonce, as per [RFC 2481] and [WES01].
- Path MTU discovery, as per [RFC 1191].

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

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

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

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

2. Design Rationale

One of the motivations behind the design of DCP is to make DCP as low-overhead as possible, in terms both of the size of the packet header and in terms of the state and CPU overhead required at the end hosts. In particular, DCP is designed to minimize the state maintained by the data sender. DCP is intended to be used by applications that currently now use UDP without end-to-end congestion control. The desire is for many applications to have little reason not to use DCP instead of UDP, once DCP is deployed.

This desire for minimal overhead results in the design decision to add only the minimal necessary functionality to DCP, and to leave other functionality such as FEC or semi-reliability to the application, to be layered on top of DCP as desired. The desire for minimal overhead is also one of the reasons to propose DCP instead of just proposing an unreliable version of SCTP for applications currently using UDP.

Mechanisms for multi-homing and mobility are the one area of additional functionality that can not necessarily be layered cleanly and effectively on top of DCP. Thus, the one outstanding design decision with DCP concerns whether to incorporate mechanisms for multi-homing and mobility into DCP itself.

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

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

3. Concepts and Terminology

3.1. Anatomy of a DCP Connection

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

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

(2) Acknowledgements from DCP B to DCP A.

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

(4) Acknowledgements from DCP A to DCP B.

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

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

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

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

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

3.2. Congestion Control

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

Conformant congestion control mechanisms correspond to single-byte congestion control identifiers, or CCIDs. The CCID for a half-connection describes how the HC-Sender limits data packet rates in a TCP-friendly manner; how it maintains necessary parameters, such as congestion windows; how the HC-Receiver sends congestion feedback via acknowledgements; and how it manages the acknowledgement rate. Section 6 introduces the currently allocated CCIDs, which are defined in separate profile documents.

The special CCID 0, Single-Window Congestion Control [CCID 0 PROFILE], is reserved for half-connections containing at most an initial window's worth of data. (The initial window is defined as in TCP; it is currently 2 packets.) This is useful for scenarios such as broadcast media, where all data travels from a "server" to a "client". If the client-to-server half-connection uses CCID 0, the server may use a simplified DCP implementation -- for instance, it need not keep lots of information about acknowledgements. We have not yet determined whether CCID 0 should reliably transmit this initial window of packets.

3.3. Connection Initiation and Termination

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

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

DCP also does not support half-open connections. That is, DCP shuts down both half-connections as a unit.

3.4. Features

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

The Ask, Choose, and Answer options negotiate feature values. Ask is sent to a feature location, asking it to change its value for the feature. The feature location may respond with Choose, which asks the other endpoint to Ask again with different values, or it may change the feature value and acknowledge the request with Answer. Retransmissions make feature negotiation reliable. Section 5.3 describes these options further.

4. DCP Packets

DCP has nine different packet types:

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

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

The progress of a typical DCP connection is as follows.

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

(2) The server sends the client a DCP-Response packet indicating that it is willing to communicate with the client. The response indicates any features and options that the server agrees to, whether an application request in the DCP-request was actually passed to the application, and optionally 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 DCP-Ack packet that acknowledges the DCP-Response packet. This acknowledges the server's initial sequence number and returns the Init Cookie if there was one in the DCP-Response. It may also continue feature negotiation.

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

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

server will send DCP-Data and DCP-DataAck packets, while the client will send DCP-Acks exclusively.

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

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

(8) The server sends a DCP-Reset packet and clears its connection state.

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

An alternative connection closedown sequence is initiated by the client:

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

(7) The server sends a DCP-Reset packet and clears its connection state.

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

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

4.1. Examples of DCP Congestion Control

Before giving the detailed specifications of DCP, we first give two more detailed examples on DCP congestion control in operation.

4.1.1. DCP with TCP-like Congestion Control

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

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

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

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

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

Some of these data packets are DCP-DataAck packets acknowledging data and/or ack packets from the client.

(5) The client sends a DCP-Ack packet acknowledging the data packets for every Ack Ratio data packets transmitted by the server. Each DCP-Ack packet uses a sequence number and contains an Ack Vector, as defined in Section 7 on Acknowledgements.

These packets also include Answer options answering any Ack Ratio requests from the server.

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

(7) Because DCP-Ack packets use sequence numbers, the server has direct information about the fraction of loss or marked DCP-Ack packets. The server responds to lost or marked DCP-Ack packets by modifying the Ack Ratio sent to the client, as described in [CCID 2 PROFILE].

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

(9) Each DCP-Data, DCP-DataAck, and DCP-Ack packet is sent as ECN-Capable, with either the ECT(0) or the ECT(1) codepoint set, as described in [WES01]. The client echoes the accumulated ECN Nonce for the server's packets along with its Ack Vector options.

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

4.1.2. DCP with TFRC Congestion Control

This example is of a connection where both half-connections use TFRC Congestion Control, specified by CCID 3 The specification for CCID 3 is in a separate profile [CCID 3 PROFILE]; the purpose of this example is to illustrate the range of uses for DCP.

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

(2) The server sends DCP-Data packets, where the number of packets sent is governed by an allowed transmit rate, as in TFRC. The details of the allowed transmit rate are defined in the profile for CCID 3, which is a separate document [CCID 3 PROFILE]. Each DCP-Data packet has a sequence number, a timestamp, the server's estimate of the round-trip time, and the current sending rate.

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

(3) The client sends DCP-Ack packets at most once per round-trip time, or as indicated by the Ack Ratio, acknowledging the data packets. These acknowledgements may be piggybacked on data packets, producing DCP-DataAck packets. Each DCP-Ack packet uses a sequence number and identifies the most recent packet received from the server, a timestamp, and feedback about the loss event rate calculated by the client, as specified by [CCID 3 PROFILE].

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

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

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

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

4.2. DCP Generic Packet Header

All DCP packets begin with a generic DCP packet header:

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

Source and Destination Ports: 16 bits each

These fields identify the connection. Packets sent on the other sequence switch the source and destination port values.

Type: 4 bits

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

0 DCP-Request packet.

1 DCP-Response packet.

2 DCP-Data packet.

3 DCP-Ack packet.

4 DCP-DataAck packet.

5 DCP-CloseReq packet.

6 DCP-Close packet.

7 DCP-Reset packet.

8 DCP-Move packet.

Reserved (Res): 4 bits

This field is reserved for future expansion. The version of DCP specified here MUST set the field to all zeroes on generated packets, and ignore its value on received packets.

Sequence Number: 24 bits

The sequence number field is initialized by a DCP-Request or DCP-Response packet, and increases by one (modulo 16777216) with every packet sent. The receiver uses this information to determine whether packet losses have occurred. Even packets containing no data update the sequence number. Sequence numbers also provide some protection against old and malicious packets. Section 4.3 discusses sequence number validity.

Data Offset: 8 bits

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

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

DCP sets this field to the number of non-data packets it has sent so far on its sequence, modulo 16. A non-data packet is simply any packet not containing user data; Data-Ack packets are the canonical example. When sending a non-data packet, DCP increments the # NDP counter before storing its value in the packet header.

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

Checksum Length (Cslen): 4 bits

The checksum length field specifies how much of the packet (in 32-bit words) following the DCP Options is covered by the checksum. If this field is 15, the entire packet is covered by the checksum. If this field is zero, only the DCP header and options are covered by the checksum. By setting the checksum length field to a value other than 15, a sender specifies that corruption is acceptable in some of the DCP packet's payload, and that partially corrupted data packets may be received and counted for congestion control purposes. For this field to be meaningful when set to a value other than 15, the link-layer must also support selective CRC mechanisms.

Checksum: 16 bits

DCP uses the TCP/IP checksum algorithm. Specifically, the checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the DCP header and options and, depending on the value of the checksum length field, some or all of the payload. When calculating the checksum, the checksum field itself is treated as 0. If a packet contains an odd number of header and text octets to be checksummed, the last octet is padded on the right with zeros to form a 16 bit word for checksum purposes. The pad is not transmitted as part of the packet.

4.3. Sequence Number Validity

DCP should ignore packets with invalid sequence numbers, which may arise if the network delivers a very old packet or an attacker attempts to hijack a connection. TCP solves this problem with its window. In DCP, however, the definition of "unreasonable sequence number" is complicated because sequence numbers change with each packet sent. Thus, a loss event that dropped many consecutive packets could cause two DCPs to get out of sync relative to any window.

This issue requires further research, but we believe that any solution will involve some combination of three mechanisms:

(1) The definition of a "loss window", with a reasonable default, that may or may not be negotiable by the DCP endpoints.

(2) Packets whose sequence numbers are beyond one loss window in the future are ignored, *unless* their acknowledgement numbers are correct (within a loss window of the last packets sent from the receiver).

(3) The use of a nonce, negotiated at connection setup time, for resyncing the remaining case of simulatenous, large numbers of consecutive losses on both packet sequences.

Any packet with an invalid sequence number (and an invalid acknowledgement number) SHOULD be ignored by the receiving DCP---that is, it should not pass any enclosed data to the application, update its congestion control state, or close the connection. However, the receiving DCP MAY send a DCP-Ack packet to the sender, as allowed by the congestion control mechanism in use. This packet should contain the last received valid sequence number. The other DCP can use the sequence number on the DCP-Ack to resync. We note that this mechanism also needs further research.

4.4. DCP-Request Packet Format

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                      Generic DCP Header                       /
/                          (12 octets)                          /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Service Name                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
|                              ...                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The Service Name field, in combination with the Destination Port, identifies the service to which the sender is trying to connect. Service Names are 32-bit numbers allocated by the IETF; they are meant to correspond to application services and protocols. The host operating system MAY force every DCP socket, both actively and passively opened, to specify a Service Name. The connection will succeed only if the Destination Port on the receiver has the same Service Name as that given in the packet. If they differ, the receiver will respond with a DCP-Reset packet.

The DCP-Request packet initializes the client-to-server sequence number. As in TCP, this sequence number should be chosen randomly to help prevent connection hijacking.

4.5. DCP-Response Packet Format

In the second phase of the three-way handshake, the server sends a DCP-Response message to the client. The response initializes the server-to-client sequence number. As in TCP, this sequence number should be chosen randomly to help prevent connection hijacking.

In this phase, a server will often specify the options it would like to use, either from among those the client requested, or in addition to those. Among these options is the congestion control mechanism the server expects to use.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                       Generic DCP Header                      /
/                          (12 octets)                          /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Reserved    |            Acknowledgement Number             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              data                             |
|                               ...                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Acknowledgement Number: 24 bits

The acknowledgement number field acknowledges the *largest* valid sequence number received so far on this connection. (The usual care must be taken in case of wrapped sequence numbers.) In the case of a DCP-Response packet, the acknowledgement number field will equal the sequence number from the DCP-Request. Acknowledgement numbers make no attempt to provide precise information about which packets have arrived; options such as the Ack Vector do this.

4.6. DCP-Data, DCP-Ack, and DCP-DataAck Packet Formats

The payload data in a DCP connection is sent in DCP-Data and DCP-DataAck packets. DCP-Data packets look like this:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                       Generic DCP Header                      /
/                          (12 octets)                          /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              data                             |
|                               ...                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                      Generic DCP Header                       /
/                          (12 octets)                          /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Reserved    |             Acknowledgement Number            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

DCP-DataAck packets contain both data and an acknowledgement number. That is, 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 DCP Header                       /
/                          (12 octets)                          /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Reserved    |             Acknowledgement Number            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
|                              ...                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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

DCP A sends DCP-Data and DCP-DataAck packets to DCP 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, DCP-Ack packets sent by DCP A are controlled by the CCID for the B-to-A half-connection. Generally, DCP A will piggyback acknowledgement information on data packets when acceptable, creating DCP-DataAck packets. DCP-Ack packets are used when there is no data to send from DCP A to DCP B, or when the link from A to B is completely congested (so sending data would be inappropriate).

Section 7, below, describes acknowledgements in DCP.

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

4.7. DCP-CloseReq and DCP-Close Packet Format

The DCP-CloseReq and DCP-Close packets have the same format. However, only the server can send a DCP-CloseReq packet. Either client or server may send a DCP-Close 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 DCP Header                      /
/                           (12 octets)                         /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Reserved    |            Acknowledgement Number             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Options                   |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

4.8. DCP-Reset Packet Format

DCP-Reset packets unconditionally shut down a connection. Every connection shutdown sequence ends with a DCP-Reset, but resets may be sent for other reasons, including bad port numbers, bad option behavior, incorrect ECN Nonce Echoes, and so forth. The reason for a reset is represented in the reset itself by a four-byte number, the Reason field.

```
 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 DCP Header                      /
/                           (12 octets)                         /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Reserved    |            Acknowledgement Number             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Reason                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Options                   |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Reason: 32 bits

The Reason field represents the reason that the sender reset the DCP connection. Particular values for this field will be described in later versions of this document.

4.9. DCP-Move Packet Format

The DCP-Move packet type is part of DCP's support for multihoming and mobility, which is described further in Section 9. DCP A sends a DCP-Move packet to DCP B after changing its IP address and/or port number. The DCP-Move packet requests that DCP B start sending its data to the new address and port number. The old address and port are stored explicitly in the DCP-Move packet header; the new address and port come from the IP header and generic DCP header. The Sequence Number, Acknowledgement Number, and mandatory Mobility Nonce fields provide some protection against hijacked connections. See Section 9 for more on security and DCP's mobility support.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                        Generic DCP Header                     /
/                           (12 octets)                         /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Reserved    |           Acknowledgement Number              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Old IP Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Old Port             |          Reserved             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Mobility Nonce                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
|                              ...                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Old IP Address: 32 bits
The former IP address used by DCP A's endpoint.

Old Port: 16 bits
The former port number used by DCP A's endpoint.

Mobility Nonce: 32 bits
The Mobility Nonce negotiated for use by DCP A.

DCP B should reset the connection if the DCP-Move packet has valid sequence and acknowledgement numbers, but an incorrect Mobility Nonce. Also, it should reset if neither the Old Address/Old Port combination nor the IP Address/Source Port combination refers to a currently active DCP connection.

DCP B MUST respond to the DCP-Move packet with a DCP-Ack or DCP-DataAck packet acknowledging the move. If this acknowledgement is lost, DCP A might resend the DCP-Move packet (using a new sequence number). DCP B MUST NOT reset these packets, even though the Old Address/Old Port combination no longer refers to a valid DCP connection. It SHOULD instead send another acknowledgement, as allowed by the congestion control mechanism in use.

5. Options and Features

All DCP packets can contain options which can be used to extend DCP's functionality. Options may occupy space at the end of the DCP header and are a multiple of 8 bits in length. All options are included in the checksum. An option may begin on any byte boundary.

The first octet of an option is the option type. Options with types 0 through 31 are single-byte options. Other options are followed by an octet indicating the option's length. The option-length counts the two octets of option-type and option-length as well as the option-data octets.

The following options are currently defined:

Type	Option Length	Meaning	Section Reference
0	1	Padding	5.1
1	1	Data Discarded	5.4
32	4	Ignored	5.2
33	variable	Ask	5.3
34	variable	Choose	5.3
35	variable	Answer	5.3
36	variable	Init Cookie	5.5
37	variable	Ack Vector [Nonce 0]	7.5
38	variable	Ack Vector [Nonce 1]	7.5
39	3	Receive Buffer Drops	7.6
40	6	Timestamp	5.6
41	10	Timestamp Echo	5.7
42	6	Mobility Nonce	9.2
128-255	variable	CCID-Specific Options	6.5

5.1. Padding Option

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

5.2. Ignored Option

The Ignored option, with type 32, signals that a DCP did not understand some option. This can happen, for example, when a conventional DCP converses with an extended DCP. Each Ignored option has two octets of payload, the first containing the offending option type and the second containing the first octet of the offending option's payload. (If the offending option had no payload, this octet is 0.)

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

5.3. Feature Negotiation

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

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

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

Some features are non-negotiable, meaning that the feature location MUST set its value to whatever the other endpoint requests. (The Ask option, for non-negotiable features, is more like "Command".) These features use the feature framework simply to achieve reliability.

5.3.1. Feature Numbers

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

Number	Meaning	Neg.?	Section Reference
1	Congestion Control (CC)	Y	6
2	ECN Capable	Y	8.1
3	Ack Ratio	N	7.3
4	Use Ack Vector	Y	7.4
5	Mobility Capable	Y	9.1
128-255	CCID-Specific Features	?	6.5

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

5.3.2. Ask Option

DCP B sends an Ask option to DCP A to ask it to change the value of some feature. (DCP A is the feature location.) DCP A MUST respond to the Ask option with either Choose or Answer. DCP B MUST retransmit the Ask option until it receives some relevant response. DCP B will always generate an Ask option in response to a Choose option; it may also generate an Ask option due to some application event.

5.3.3. Choose Option

DCP A sends a Choose option to DCP B to ask it to confirm the value of some feature. (Again, DCP A is the feature location.) DCP B MUST respond to the Choose option with an Ask. DCP A MUST retransmit the Choose option until it receives a relevant Ask response. DCP A may generate a Choose option in response to some Ask option, or in response to some application event.

5.3.4. Answer Option

DCP A sends an Answer option to DCP B to inform it of the current value of some feature. (Again, DCP A is the feature location.) DCP A MUST generate Answer options only in response to Ask options. DCP A need not ever retransmit an Answer option: DCP B will retransmit the relevant Ask as necessary.

5.3.5. Example Negotiations

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

```
B > A    Ask(CC, 2)
A > B    Answer(CC, 2)
```

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

```
B > A    Ask(CC, 3, 4)
A > B    Choose(CC, 1, 2, 5)
B > A    Ask(CC, 5)
A > B    Answer(CC, 5)
```

In this sequence, A refuses to use CC mechanism 5. If B requires CC mechanism 5, its only recourse is to abort the connection:

```
B > A    Ask(CC, 3, 4, 5)
A > B    Choose(CC, 1, 2)
B > A    Ask(CC, 5)
A > B    Choose(CC, 1, 2)
```

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

```
A > B    Choose(CC, 1, 2)
B > A    Ask(CC, 2)
A > B    Answer(CC, 2)
```

5.3.6. Unknown Features

If a DCP receives an Ask or Choose option referring to a feature number it does not understand, it MUST respond with a corresponding Ignored option. This informs the remote DCP that the local DCP does not implement the feature. No other action need be taken. (Ignored may also indicate that the DCP endpoint could not respond to a CCID-specific feature request because the CCID was in flux; see Section 6.5.)

5.3.7. State Diagram

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

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

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

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

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

```
                    REQUESTER STATE DIAGRAM (DCP B)

                        +-----------+
                        |  Unknown  |
                        +-----------+
      +----------+            |                    +-----------+
      |          |RECV -      |RECV -/Ch | APP     |           |RECV Ch/Ans
      V          |SEND -      |SEND Ask            V           |SEND Ask
+-----------+    |            |             +------------+     |
|           |----+            +------------>|            |-----+
|   Known   |------------------------------>|   Asking   |
|           |        RECV Ch | APP          |            |-----+
+-----------+          SEND Ask             +------------+     |RECV -
      ^                                          | | ^         |SEND -/Ask
      |                                          | | |         |
      +------------------------------------------+ | +---------+
                       RECV Ans(O)                 |          +----------+
                       SEND -                      +--------->|  Failed  |
                       SET-VALUE O                  RECV Ign  +----------+
                                                    SEND -
```

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


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

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

DCP B must retransmit Ask options, and DCP A must retransmit Choose options, until receiving a relevant response. However, they need not retransmit the option on every packet, as shown by the "`RECV - / SEND -`" transitions in the Asking and Confirming states.

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

```
A > B    Choose(1)
B > A    Ask(2)
A > B    Choose(1)
B > A    Ask(2)...
```

Implementations may choose to enforce a maximum length on any negotiation -- for example, by resetting the connection when any negotiation lasts more than some maximum time.

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

5.4. Data Discarded Option

This option is permitted in a DCP-Response packet only. It indicates that the payload of the DCP-Request packet was discarded by the server, and therefore should be resent in a following DCP-Data or DCP-DataAck packet. This option can be set by the server to avoid having to keep state for the connection until the handshake is complete. Doing so causes an additional round-trip time before the server can begin servicing the request. The tradeoff is under the control of local policy at the server.

5.5. Init Cookie Option

This option is permitted in DCP-Response, DCP-Data, and DCP-DataAck messages. The option MAY be returned by the server in a DCP-Response mechanism. If so, then the client MUST echo the same Init Cookie option in its ensuing DCP-Data or DCP-DataAck message.

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

The precise implementation of the Init Cookie does not need to be specified here as it is only relayed by the client, and does not need to be understood by the client.

5.6. Timestamp Option

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

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

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

5.7. Timestamp Echo Option

This option is permitted in any DCP packet, as long as at least one packet carrying the Timestamp option has been received. The length of the option is 10 bytes.

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

The first four bytes of option data, TS Echo, carry a Timestamp Value taken from a preceding received Timestamp option. Usually, this will be the last packet that was received. The final four bytes indicate the amount of time elapsed since receiving the packet whose timestamp is being echoed. This time MUST be in microseconds. We are currently investigating ways to relax the last requirement.

6. Congestion Control IDs

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

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

The CCIDs defined by this document are:

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

A new connection starts with CCID 0 for both DCPs. If this is unacceptable for either DCP, that DCP will start in the Unknown state. A DCP SHOULD NOT send data when its Congestion Control feature is in the Unknown state.

6.1. Single-Window Congestion Control

CCID 0 denotes the absence of congestion control, and is appropriate only for streams of pure acknowledgements, possibly including at most one window of data at connection startup. (Streams of pure acknowledgements are congestion controlled, but by the other half-connection’s CCID. See Section 7 below.) This is appropriate for half-connections that will contain no data---for example, the client-to-server half-connection on a streaming media connection. Servers may want to encourage their clients to use CCID 0, since this will ensure that they need not maintain detailed acknowledgement information for clients’ packets, simplifying their implementation.

HC-Senders using CCID 0 MUST NOT send any data packets during the lifetime of the connection, possibly after at most one initial window of data (as defined by TCP; currently two packets) during connection startup. HC-Receivers using CCID 0 SHOULD reset the connection if they receive an unexpected data packet.

We have not yet determined whether CCID 0 should reliably transmit this initial window of packets.

CCID 0 is further described in [CCID 0 PROFILE].

6.2. Unspecified Sender-Based Congestion Control

CCID 1 denotes an unspecified sender-based congestion control mechanism. Separate features negotiate the corresponding congestion acknowledgement options -- for example, Ack Vector.

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

6.3. TCP-like Congestion Control

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

6.4. TFRC Congestion Control

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

6.5. CCID-Specific Options and Features

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

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

For example, consider a DCP 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	Ask(128, ...)	B-to-A	5
A > B	Choose(128, ...)	A-to-B	4
A > B	Answer(128, ...)	A-to-B	4
A > B	Ask(192, ...)	A-to-B	4
A > B	Choose(192, ...)	B-to-A	5
A > B	Answer(192, ...)	B-to-A	5

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

7. Acknowledgements

Congestion control requires receivers to transmit information about packet losses and ECN marks to senders. DCP receivers MUST report all congestion they see, using mechanisms appropriate for the CCID in use.

Generally, this is accomplished through options. For example, on a half-connection with CCID 2 (TCP-like), the receiver reports acknowledgement information using the Ack Vector option. CCID-specific profiles say which options are relevant, and how to decide when to ack; this section describes common acknowledgement options and shows how acks using those options will commonly work. Acknowledgement options, such as Ack Vector, are only allowed on DCP-Ack, DCP-DataAck, DCP-Close, and DCP-CloseReq packets.

7.1. Acknowledgements and CCIDs

Acknowledgements are controlled by CCIDs. Each CCID specifies which options its acknowledgements must use, when they should be sent, how they should be congestion controlled, and so on. Each CCID additionally describes the form acks-of-acks must take -- if required at all -- when the CCID is active on a unidirectional connection. This last point requires some explanation.

DCP 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 bidirectional connection are very different from those required for a unidirectional connection.

Consider a connection where both connections use the same CCID (either 2 or 3), but the B-to-A half-connection has become quiescent; that is, DCP B has no more data to send to DCP A, and is sending only DCP-Acks. Now, for CCID 2, TCP-like Congestion Control, DCP B uses Ack Vector to reliably communicate which packets it has received. Because of this reliability, DCP A must inform DCP B when it receives an Ack Vector: that is, DCP A must occasionally acknowledge a pure acknowledgement. The ack-of-ack traffic need not be reliable; for instance, it need not use Ack Vector. DCP A might just send a DCP-DataAck packet every now and then, instead of DCP-Data. In contrast, for CCID 3, TFRC Congestion Control, DCP B's acknowledgements need not be reliable. B's DCP-Acks contain cumulative loss rates; TFRC works even if every DCP-Ack is lost. Therefore, DCP A need not ever acknowledge an acknowledgement.

When communication is bidirectional, DCP A's ack-of-ack traffic is automatically contained in its normal acknowledgement traffic for DCP B's data. However, the required ack-of-ack traffic is significantly smaller and simpler than the normal ack traffic. Therefore, DCP sends only the ack-of-ack traffic when communication is unidirectional, since this reduces DCP A's acknowledgements to nothing, or nearly nothing. Thus, when communication is unidirectional, a single CCID -- in the example, the A-to-B CCID -- is controlling both DCP A's and DCP B's acknowledgements, in terms of their content, their frequency, and so forth. In the bidirectional case, the A-to-B CCID governs DCP B's acknowledgements, while the B-to-A CCID governs DCP A's acknowledgements.

DCP A switches its ack pattern from bidirectional to unidirectional when it notices that DCP B has gone quiescent -- that is, B is no longer sending data packets. It switches from unidirectional to bidirectional when it must acknowledge even a single DCP-Data or DCP-DataAck packet from DCP B. (This includes the case where a single DCP-Data or DCP-DataAck packet was lost in transit. DCP A can detect this case using the # NDP field in the DCP packet header.) The B-to-A CCID defines when DCP B has gone quiescent; usually, this happens when a period has passed without B sending any data packets. For CCID 2, this period is roughly two round-trip times. The A-to-B CCID defines how DCP A handles acks-of-acks once DCP B has gone quiescent.

7.2. Ack Piggybacking

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

Piggybacking is particularly common at DCP A when the B-to-A half-connection is quiescent -- that is, when DCP A is just acknowledging DCP B's acknowledgements, as described above. There are three reasons to acknowledge DCP B's acknowledgements: to allow DCP B to free up information about previously acknowledged data packets from A; to shrink the size of future acknowledgements; and to manipulate the rate future acknowledgements are sent. Since these are secondary concerns, DCP 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.

7.3. Ack Ratio Feature

With Ack Ratio, DCP A can perform rudimentary congestion control on DCP B's acknowledgement stream by telling DCP B how to clock its acks.

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

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

This feature is non-negotiable.

7.4. Use Ack Vector Feature

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

Use Ack Vector has feature number 4. The Use Ack Vector feature located at DCP B specifies whether DCP B should use the Ack Vector option to report congestion back to DCP A. DCP A sends an "Ask(Use Ack Vector, 1)" option to DCP B to ask B to send Ack Vector options as part of its acknowledgement traffic.

A Use Ack Vector option contains a single octet of data. The receiver should send Ack Vector options if and only if this octet is nonzero. A new connection starts with Use Ack Vector 0 for both DCPs.

7.5. Ack Vector Options

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

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

The two Ack Vector options (option types 37 and 38) differ only in the values they imply for ECN Nonce Echo. Section 8.2 describes this further.

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

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

St[ate]: 2 bits

Run Length: 6 bits

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

0 Packet received (and not ECN marked).

1 Packet ECN marked.

2 Reserved.

3 Packet not yet received.

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

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

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

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

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

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

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

Packets dropped in the receive buffer should be reported as not received (State 3). The Receive Buffer Drops option distinguishes between congestion losses and losses due to receive buffer overflow.

7.5.1. Ack Vector Consistency

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

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

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

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

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

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. However, 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).

7.5.2. Ack Vector Coverage

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

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

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

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

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

The union of groups 2 and 3 is called the Unacknowledged Window. Generally, every Ack Vector the HC-Receiver sends will cover the whole Unacknowledged Window: Ack Vector acknowledgements are cumulative. (This simplifies Ack Vector maintenance at the HC-Receiver; see Section 7.7, 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.

7.6. Receive Buffer Drops Option

The Receive Buffer Drops option indicates that some packets reported as not received, were actually dropped at the endpoint due to insufficient kernel space. The sender will probably react differently to receive buffer drops than congestion losses; for instance, it might not reduce its congestion window. The option's data looks like this:

```
+--------+--------+--------+
|00100111|00000011| Count  |
+--------+--------+--------+
 Type=39  Length=3
```

Count: 8 bits

The Count field says how many acknowledged packets were dropped at the receive buffer, limited to packets acknowledged by the packet containing the option. Count is simply a number between 0 and 255.

Multiple Receive Buffer Drops options are added together, so a single option with Count 2 is equivalent to two options, each with Count 1. A packet's total Receive Buffer Drops count MUST be less than or equal to the number of packets acknowledged by it as "not yet received". For example, assuming Ack Vector, the Receive Buffer Drops count must be less than or equal to the total number of State-3 packets in the Ack Vectors.

If an ECN-marked packet is dropped at the receive buffer, it MUST NOT be included in the Receive Buffer Drops count. Such packets MUST be reported as the equivalent of "dropped by the network". (For Ack Vector, this is "not yet received".)

7.7. Ack Vector Implementation Notes

This section discusses the particulars of DCP acknowledgement handling, in the context of an abstract implementation for Ack Vector. It may safely be skipped.

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

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

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

```
+-------------------------------------------------------------------+
|S,L|S,L|S,L|S,L|S,L|   |   |   |   |S,L|S,L|S,L|S,L|S,L|S,L|S,L|S,L|
+-------------------------------------------------------------------+
                  ^                   ^
            Tail, seqno = T     Head, seqno = H

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

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

```
         +----------------------------------------------------+
(Head) H |S,L|S,L|S,L|S,L|S,L|S,L|S,L|S,L|S,L|S,L|S,L|S,L|S,L| T (Tail)
         +----------------------------------------------------+
```

This smaller Example Buffer contains actual data.

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

7.7.1. New Packets

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

```
          +***----------------------------+
       11 |1,0|0,0|3,0|3,0|3,0|0,4|1,0|0,0|  0
          +***----------------------------+
```

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, the Example Buffer might enter this state instead:

```
              +--*------------------------+
           11 |0,1|3,0|3,0|3,0|0,4|1,0|0,0|  0
              +--*------------------------+
```

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

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

```
              +--------*------------------+
           10 |0,0|3,0|0,0|3,0|0,4|1,0|0,0|  0
              +--------*------------------+
```

If the packet is not marked as lost, or if its sequence number is not contained in the table, the packet is probably a duplicate, and should be ignored. (The new packet's ECN marking state might differ from the state in the buffer; Section 7.5.1 describes what to do then.) If the packet's corresponding buffer byte has a non-zero run length, then the buffer might need be reshuffled to make space for one or two new bytes.

Of course, the circular buffer may overflow, either when the HC-Sender is sending data at a very high rate, when the HC-Receiver's acknowledgements are not reaching the HC-Sender, or when the HC-Sender is forgetting to acknowledge those acks (so the HC-Receiver is unable to clean up old state). In this case, the HC-Receiver should either compress the buffer, transfer its state to a larger buffer, or drop all received packets until its buffer shrinks again.

7.7.2. Sending Acknowledgements

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

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

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

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

7.7.3. Clearing State

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

```
HC-Receiver Ack 59  acknowledged  HC-Sender Seq 3, and
HC-Receiver Ack 60  acknowledged  HC-Sender Seq 10.
```

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

```
   +------------------*+  *
10 |0,0|3,0|3,0|3,0|0,2|  4
   +------------------*+  *
```

Note that the tail byte's run length was adjusted, since packet 3 was in the middle of that byte. The HC-Receiver can also throw away the information about HC-Receiver Ack 59.

A careful implementation might also modify its own acknowledgement record to ensure that it is reasonably robust to reordering. Suppose that the Example Buffer is as before, but that packet 9 now arrives, out of sequence. The Example buffer would enter this state:

```
   +----*----------------------+
10 |0,0|0,0|3,0|3,0|0,4|1,0|0,0|  0
   +----*----------------------+
```

Now, if the HC-Receiver then received a DCP-DataAck packet from the HC-Sender with Sequence Number 11 and Acknowledgement Number 60, this might cause the tail pointer to be moved up to packet 10, although packet 9's arrival has not yet been acknowledged. Instead, when packet 9 arrived, the HC-Receiver's acknowledgement record might be modified to:

```
HC-Receiver Ack 59  acknowledged  HC-Sender Seq 3, and
HC-Receiver Ack 60  acknowledged  HC-Sender Seq 8.
```

That is, any HC-Sender sequence number in the acknowledgement record is reduced to at most 8. 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 this packets arrival.

7.7.4. Processing Acknowledgements

When the HC-Sender receives an acknowledgement, it generally cares about the number of packets that were dropped and/or ECN marked. It simply reads this off the Ack Vector. Additionally, it may check the ECN Nonce for correctness. (As described in Section 7.5.1, it may want to keep more detailed information about acknowledged packets in case packets change states between acknowledgements, or in case the application queries whether a packet arrived.)

Of course, the HC-Sender must also acknowledge the HC-Receiver's acknowledgements, so the HC-Receiver can free up its state. This is much simpler than the HC-Receiver's acknowledgement code, since the HC-Receiver doesn't need complete acknowledgement information. For example, assuming that the HC-Receiver sends no data, the HC-Sender can simply ensure that at least once a round-trip time, it sends a DCP-DataAck packet acknowledging the latest DCP-Ack packet it has received. (The HC-Sender must watch for drops and ECN marks on received DCP-Ack packets, so that it can adjust the HC-Receiver's ack-sending rate in response to congestion; but it need not inform the HC-Receiver about which acks were dropped.)

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 usually sufficient for the HC-Receiver to free its state.

8. Explicit Congestion Notification

The DCP protocol is fully ECN-aware. Every CCID specifies how its endpoints respond to ECN marks. Furthermore, DCP, 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.

Every CCID profile describes how that profile interacts with ECN, both for data traffic and pure-acknowledgement traffic. A sender SHOULD set ECN-Capable Transport on a sent packet whenever the receiver has its ECN Capable feature turned on, and the relevant CCID allows it.

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

8.1. ECN Capable Feature

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

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

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

A new connection starts with ECN Capable 1 (that is, ECN capable) for both DCPs. If a DCP is not ECN capable, it MUST send "Choose(ECN Capable, 0)" options to the other endpoint until acknowledged (by "Ask(ECN Capable, 0)") or the connection closes. Furthermore, it MUST NOT accept any data until the other endpoint sends "Ask(ECN Capable, 0)".

8.2. ECN Nonces

Congestion avoidance will not occur, and the receiver will sometimes get its data faster, when the sender is not told about any congestion events. Thus, the receiver has some incentive to falsify acknowledgement information, reporting that marked or dropped packets were actually received unmarked. This problem is more serious with DCP 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 DCP, the ECN Nonce Echo field is encoded in acknowledgement options. For example, the Ack Vector option comes in two forms, Ack Vector [Nonce 0] (option 37) and Ack Vector [Nonce 1] (option 38), corresponding to the two values for a one-bit ECN Nonce Echo. The Nonce Echo for a given Ack Vector equals the base-2 modulus of the number of received ECN Nonce packets represented by that Ack Vector. Only packets marked as State 0 matter for this calculation (that is, received packets that were not ECN marked or dropped in the receive buffer). 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 ECN Nonce packet, using a separate code point for received ECN Nonces. However, this would limit Ack Vector's compressibility without providing much extra protection.)

Consider a half-connection from DCP A to DCP B. DCP A SHOULD set ECN Nonces on its packets, and remember which packets had nonces, whenever DCP 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.

9. Multihoming and Mobility

DCP provides primitive support for multihoming and mobility, via a mechanism for transferring a connection endpoint from one IP address to another. The moving endpoint must negotiate mobility support beforehand; as part of this negotiation, it will receive a nonce from the stationary endpoint. When the moving endpoint gets a new IP address, it sends a DCP-Move packet from that address to the stationary endpoint, including the nonce. The stationary endpoint then changes its connection state to use the new IP address.

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

9.1. Mobility Capable Feature

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

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

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

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

Any packet containing an "Answer(Mobility Capable, 1)" option MUST also contain a Mobility Nonce option.

9.2. Mobility Nonce Option

This option is permitted only in DCP packets also containing an "Answer(Mobility Capable, 1)" option. The length of the option is 6 bytes.

```
+--------+--------+--------+--------+--------+--------+
|00101010|00000110|          Mobility Nonce           |
+--------+--------+--------+--------+--------+--------+
 Type=42  Length=6
```

The four bytes of option data carry the mobility nonce for a DCP endpoint. The nonce sent from DCP A to DCP B is a 32-bit number that must be echoed in any DCP-Move packet from B, providing a small amount of security against hijacked connections. The Mobility Nonce option is only meaningful on packets that close a Mobility Capable feature negotiation. Mobility Nonces sent at any other time MUST be ignored. If DCP A would like to choose a different Mobility Nonce, it should send a "Choose(Mobility Capable, 1)" option to DCP B. This reopens the feature negotiation; A may send the new nonce when that negotiation closes with "Answer(Mobility Capable, 1)". If DCP B would like to get a different Mobility Nonce, it should likewise reopen the feature negotiation by sending an "Ask(Mobility Capable, 1)" option to DCP A.

9.3. Security

The DCP mobility mechanism, like DCP in general, does not provide cryptographic security guarantees. Nevertheless, DCP-Move packets must have valid sequence numbers and a valid Mobility Nonce, providing protection against some classes of attackers. Specifically, an attacker cannot move a DCP connection to a new IP address unless they know both the Mobility Nonce and a valid sequence number. If initial sequence numbers and Mobility Nonces are chosen well (that is, randomly), this means that attackers must snoop on data packets to get any reasonable probability of success. Section 14 further describes DCP security considerations.

9.4. Congestion Control State

Once an endpoint has transitioned to a new IP 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 DCPs MUST initialize their congestion control state (windows, rates, and so forth) to that of a new connection---that is, they must "slow start"---unless they have high-quality information about actual network conditions between the two new endpoints. Normally, the only way to get this information would be by instrumenting a DCP connection between the new addresses.

Similarly, the endpoints' configured MTUs (see 10) should be reinitialized, and PMTU discovery performed again, following an IP address change.

9.5. Loss During Transition

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

To protect against lost DCP-Move packets, the mobile host SHOULD retransmit a DCP-Move packet if it does not receive an acknowledgement within a reasonable time period. Section 4.9 describes the mechanism used to protect against duplicate DCP-Move packets.

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

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

10. Path MTU Discovery

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

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

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

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

As specified in [RFC 1191], when a router receives a packet with DF set that is larger than the PMTU, it sends an ICMP Destination Unreachable message to the source of the datagram with the Code indicating "fragmentation needed and DF set" (also known as a "Datagram Too Big" message). When a DCP 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 DCP endpoints behind certain firewalls.)

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

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

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

11. Abstract API

TBA

12. Multiplexing Issues

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

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

13. DCP and RTP

This section discusses the relationship between DCP and RTP [RFC 1889].

TBA

14. Security Considerations

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

Nevertheless, DCP is intended to protect against some classes of attackers. Attackers cannot hijack a DCP 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 DCP attacker must snoop on data packets to get any reasonable probability of success. The sequence number validity (Section 4.3) and mobility (Section 9) mechanisms provide this guarantee.

This section is not in its final state. Further research is needed to ensure that we have met our stated security requirement.

15. IANA Considerations

DCP introduces five sets of numbers whose values should be allocated by IANA.

- 32-bit Service Names (Section 4.4).
- 32-bit DCP-Reset Reasons (Section 4.8).
- 8-bit DCP Option Types (Section 5). The CCID-specific options 128 through 255 need not be allocated by IANA.
- 8-bit DCP Feature Numbers (Section 5.3). The CCID-specific features 128 through 255 need not be allocated by IANA.
- 8-bit DCP Congestion Control Identifiers (CCIDs) (Section 6).

In addition, DCP would require a Protocol Number to be added to the registry of Assigned Internet Protocol Numbers.

16. Thanks

There is a wealth of work in this area, including the Congestion Manager. We thank the staff and interns of ACIRI, the members of the End-to-End Research Group, and the members of the Transport Area Working Group for their feedback on DCP.

17. References

[CCID 0 PROFILE] E. Kohler. Profile for DCP Congestion Control ID 0: Single-Window Congestion Control. Work in progress.

[CCID 2 PROFILE] S. Floyd, E. Kohler. Profile for DCP Congestion Control ID 2: TCP-like Congestion Control. Work in progress.

[CCID 3 PROFILE] J. Padhye. Profile for DCP Congestion Control ID 3: TFRC Congestion Control. Work in progress.

[RFC 1191] J. C. Mogul, S. E. Deering. Path MTU discovery. RFC 1191.

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

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

[RFC 2481] K. Ramakrishnan, S. Floyd. A Proposal to add Explicit Congestion Notification (ECN) to IP. RFC 2481.

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

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

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

[WES01] David Wetherall, David Ely, Neil Spring. Robust ECN Signaling with Nonces. draft-ietf-tsvwg-tcp-nonce-00.txt, work in progress, January 2001.

18. Authors’ Addresses

Eddie Kohler <kohler@aciri.org>
Mark Handley <mjh@aciri.org>
Sally Floyd <floyd@aciri.org>
Jitendra Padhye <padhye@aciri.org>

AT&T Center for Internet Research at ICSI (ACIRI),
ICSI,
1947 Center Street, Suite 600
Berkeley, CA 94704.