

Datagram Congestion Control Protocol (DCCP): Overview



Eddie Kohler
International Computer Science Institute
IETF 57 APPAREA Meeting
July 14, 2003

DCCP is



- A congestion-controlled, unreliable flow of datagrams
- “UDP plus congestion control”

Target applications

- Long-lived flows that prefer timeliness to reliability
Streaming media, Internet telephony, on-line games, ...
- TCP inappropriate, UDP often inappropriate
TCP can introduce arbitrary retransmission delay
UDP not congestion controlled, apps must implement CC
- Apps want
Buffering control: don't deliver old data
Different congestion control mechanisms (TCP vs. TFRC)
Middlebox traversal
Low per-packet byte overhead

DCCP's attractions for applications

- Congestion control implementation
 - Experience shows CC is difficult to get right
- Explicit connection setup and teardown (firewall-friendly)
- Integrated acknowledgements, reliable feature negotiation
- Access to ECN
 - ECN capable flows must be congestion controlled
 - UDP APIs would find this difficult to enforce
- Partial checksums
 - Deliver corrupt data rather than drop it
- DoS protection
- Different congestion control mechanisms →

TCP-like vs. TFRC congestion control

- TCP-like: quickly get available B/W

Cost: sawtooth rate—halve rate on single congestion event

May be more appropriate for on-line games

More bandwidth means more precise location information; UI cost of whipsawing rates not so bad

- TFRC [RFC 3448]: respond gradually to congestion

Single congestion event does not halve rate

Cost: respond gradually to available B/W as well

May be more appropriate for telephony, streaming media

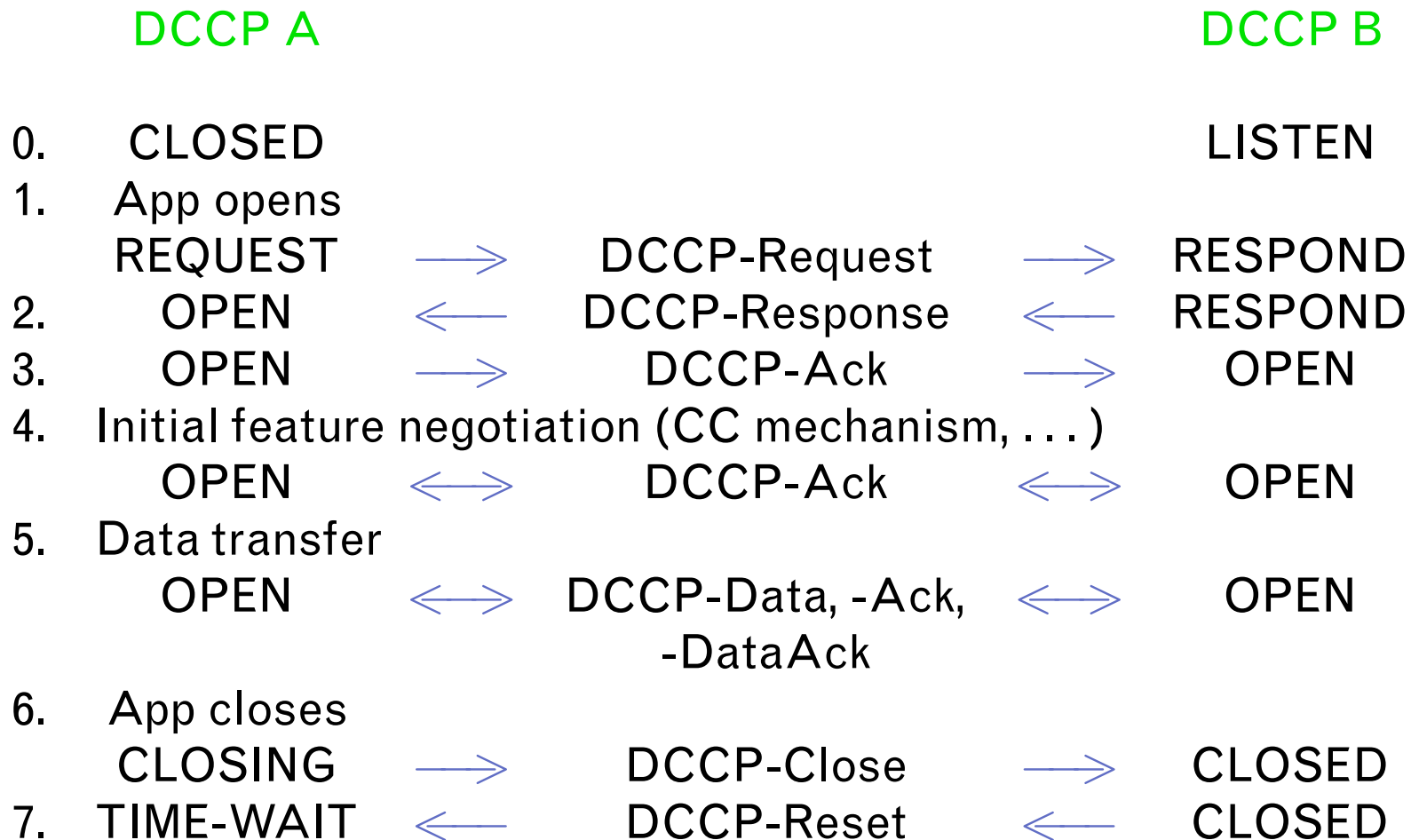
UI cost of whipsawing rates catastrophic

- DCCP will provide access to other CC mechanisms as they are standardized (TFRC-PS, ...)

DCCP's problems for applications

- App loses control over exactly when packets may be sent
 - Inherent in congestion control
 - APIs should allow late decision of what to send
- Some overhead over UDP
 - At minimum, 4 bytes per packet
 - Analysis of RTP shows minimum is often achievable
- Not yet deployed (duh)

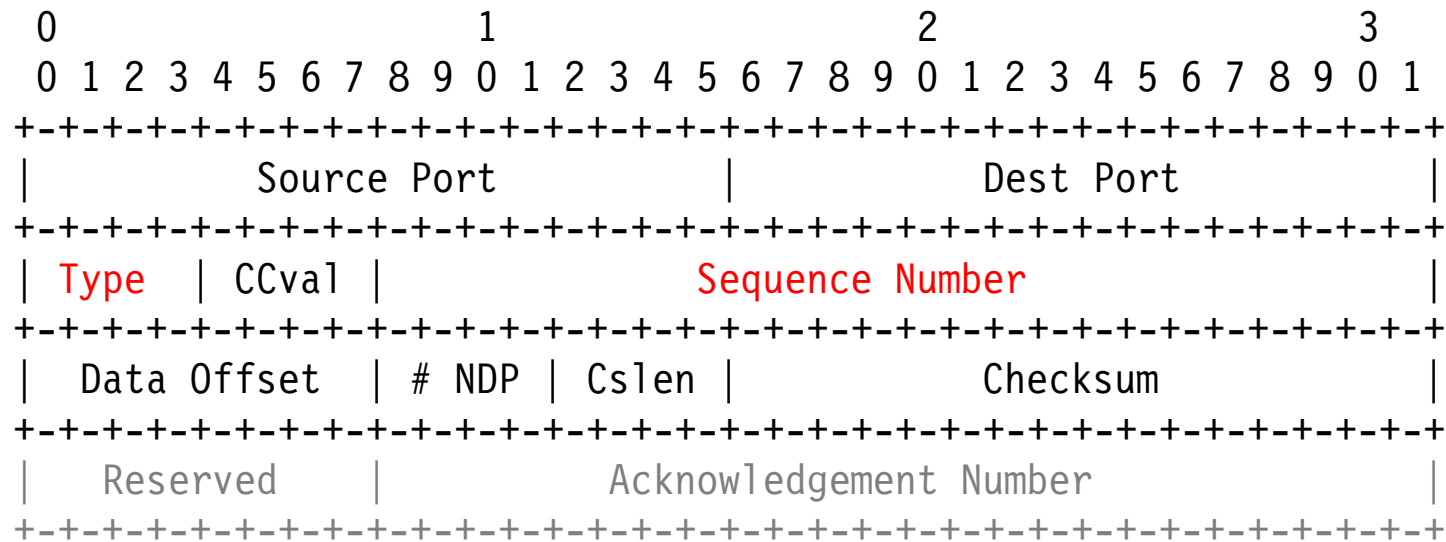
Sample connection



Two half-connections

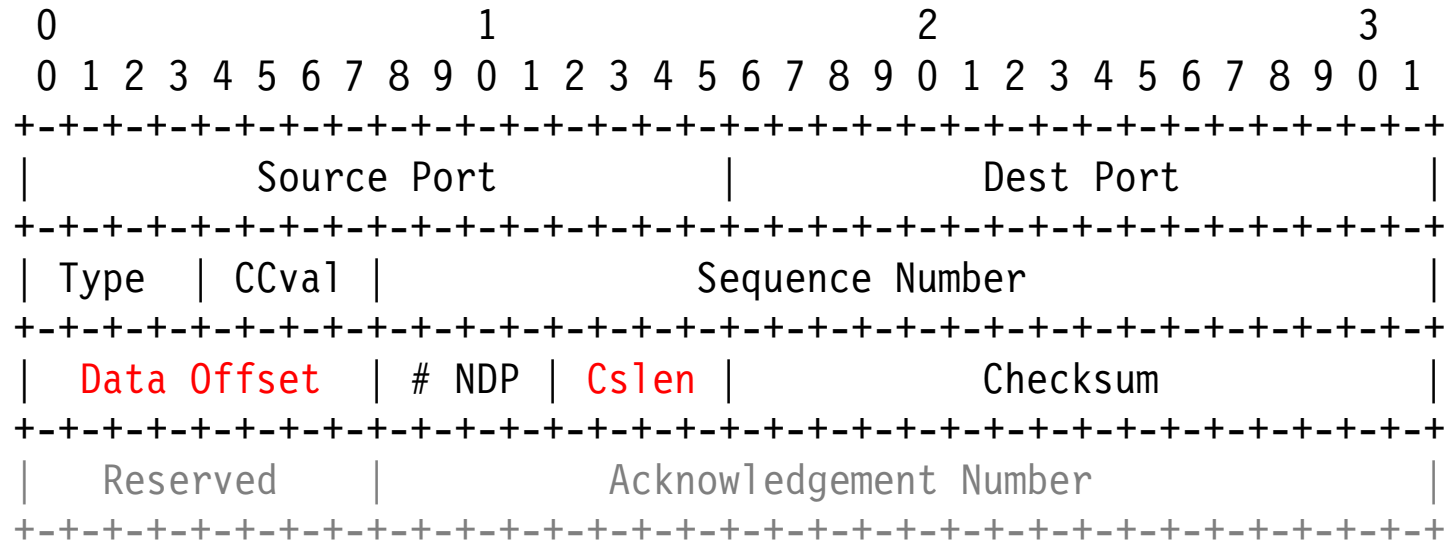
- A **half-connection** is data flowing in one direction, plus the corresponding acknowledgements
- A DCCP connection contains two half-connections
 - A \longrightarrow B data plus B \longrightarrow A acks
 - B \longrightarrow A data plus A \longrightarrow B acks
 - Can piggyback acks on data (DCCP-DataAck packet type)
- Conceptually separate
 - May use different congestion control mechanisms
 - Will this be useful for apps?
- Quiescence
 - Fewer acknowledgements for inactive half-connections

Packet header



- Sequence Number measured in packets, not bytes
Changes on every packet, even pure acks
- Gray portion not on all packet types
Different headers for different packet types (unlike TCP)
Reduce byte overhead for unidirectional connections

Packet header (2)



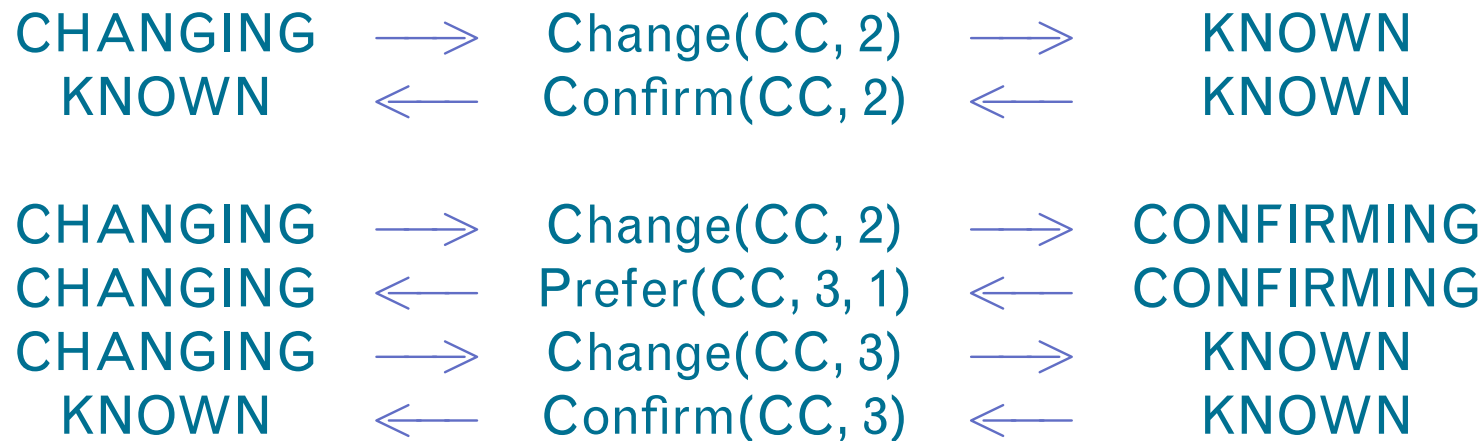
- Cslen supports partial checksums
 - Errors in header result in packet drop
 - Errors in payload, outside Cslen coverage, ignored
- Data Offset (header size in 32-bit words) leaves lots of space for options

Reliable feature negotiation

- Three options: Change, Prefer, Confirm
 - Change: “Please use this value for a feature”
 - Prefer: “I would rather use one of these values”
 - Confirm: “OK, I am using this value”
- Examples: agreeing on B’s congestion control mechanism

DCCP A

DCCP B



Ack Vector option

- Run-length encoded history of data packets received

Cumulative ack not appropriate for an unreliable protocol

Steroidal SACK

| | |
|---|-----------------------|
| +-----+-----+-----+-----+-----+----- | States (SS) |
| 001001?? Length SSLLLLLL SSLLLLLL SSLLLLLL ... | 0 received non-marked |
| +-----+-----+-----+-----+-----+----- | 1 received ECN marked |
| Type=37/38 _____ Vector _____... | 3 not yet received |

Up to 16192 data packets acknowledged per option

Includes ECN nonce

- Want API to provide Ack Vector information to app

Data Dropped option

- Ack Vector says whether a packet's *header* was processed
 Not whether packet's data will be delivered to application
 Supports drop-from-head receive buffers, ...
- Data Dropped says whether a packet's data was delivered
 And if not, why not
 Enables richer [non-]congestion response functions

```

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

```

```

 0 1 2 3 4 5 6 7
+--+--+--+--+--+--+
|0| Run Length |
+--+--+--+--+--+--+
Normal Block

```

```

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

```

Drop States

- 0 protocol constraints
- 1 receive buffer
- 2 corrupted
- 3 delivered corrupt
- 4 app not listening

APIs

- Amenable to a more-or-less conventional socket API

Socket options induce feature negotiations, report CC state

- High-performance send API

Goals: high throughput, late decision on what to send, ack information

Currently investigating ring buffer model (Junwen Lai)

App allocates ring buffer from kernel, writes packets into buffer

Kernel reads from buffer asynchronously, writes information about sent and acknowledged packets

App can remove old packets from ring buffer if it gets too far ahead

Receive analogue?

Conclusion

- <http://www.icir.org/kohler/dccp/>
 - draft-ietf-dccp-problem-01.txt: Problem Statement
 - draft-ietf-dccp-spec-04.txt: main specification
 - draft-ietf-dccp-ccid{2,3}-03.txt: CCID specs
- Design review Wednesday
- Appreciate comments from app community