DCCP
(Datagram Congestion Control Protocol)

Keith Briggs

Keith.Briggs@bt.com

`research.btexact.com/teralab/keithbriggs.html`

CRG meeting 2003 Nov 21 (should have been 17) 15:00

typeset 2003 November 21 10:04 in pdfLaTeX on a linux system

In the wireless environment

- ★ In fixed networks, packet losses are usually due to network congestion
- ★ Changes in data rate often enough to improve QoS
- ★ In wireless networks many other factors: access to radio link, interference, fading etc
- ★ Difficult to offer guarantees - continuously changing conditions

Proposal

- ★ Adapt applications using:
- ★ Codec, codec specific parameters (eg quality factor for MJPEG), frame sizes, frame rates etc
- ★ Work with Agora System's ISABEL Light real-time video conferencing application
- ★ Develop application with Agora Systems and implement into the QoS testbed
- ★ Interface application with DCCP

ISABEL Light

- ★ Developed by Agora Systems under the BRAIN & MIND EU projects
- ★ Records lost packets using RTP numbering - decides whether to upgrade/downgrade codec
- ★ Adaptation steps manually configured - audio & video codec, video size, frame rate and quality

Demonstration

- ★ ISABEL Light application running between two laptops with web
  cameras over peer-peer WLAN link
- ★ Adaptation steps configured to suit properties of link
- ★ Traffic generated on link to force adaptation

```
AUDIO: PCM VIDEO: MJPEG (40F 25 50)   1171
AUDIO: PCM VIDEO: MJPEG (43F 1)       1171
```

```
Upgrade : 5 NiceReports
Upgrade : 5 NiceReports
Upgrade : 5 NiceReports
Upgrade : 5 NiceReports
```

What is DCCP?

- ★ Datagram Congestion Control Protocol (*DCCP*) provides features including unreliable flow of datagrams with acknowledgements, reliable handshake for connection setup and teardown, together with other features
- ★ It 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

Interface with DCCP

- ⋆ Translation link being written to run ISABEL Light application over DCCP instead of UDP
- ⋆ Using kernel-based implementation of DCCP
- ⋆ Unlikely to be large improvement gains since ISABEL Light application will not be aware of new DCCP layer and therefore cannot take advantage if it

DCCP features

- ★ An unreliable flow of datagrams, with acknowledgements
- ★ Reliable handshake for connection setup and teardown
- ★ Reliable negotiation of features
- ★ A choice of TCP-friendly congestion control mechanisms, including TCP-like congestion control (CCID 2) and TCP-Friendly Rate Control (CCID 3). CCID 2 uses a version of TCP's congestion control mechanisms, and is appropriate for flows that want to quickly take advantage of available bandwidth, and can cope with quickly changing send rates; CCID 3 is appropriate for flows that require a steadier send rate
- ★ Options that tell the sender, with high reliability, which packets reached the receiver, and whether those packets were ECN marked, corrupted, or dropped in the receive buffer

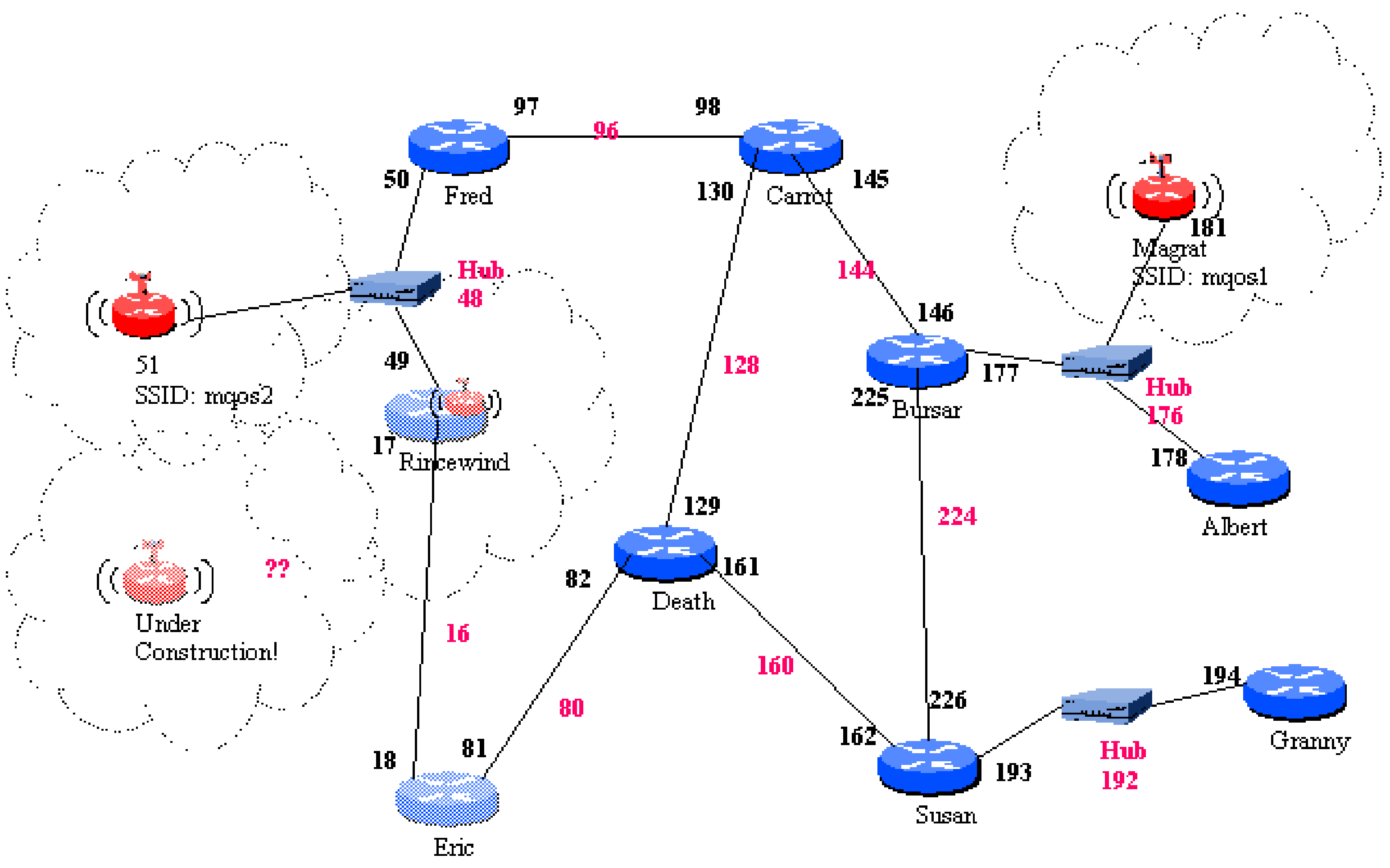
Differences from TCP

- DCCP is a packet stream protocol, not a byte stream protocol. The application is responsible for framing
- Unreliability: DCCP will never retransmit a datagram. Options are retransmitted as required to make feature negotiation and ack information reliable
- Packet sequence numbers. Sequence numbers refer to packets, not bytes. Every packet sent by a DCCP endpoint gets a new sequence number
- Choice of congestion control. One such feature is the congestion control mechanism to use for the connection. In fact, the two endpoints can use different congestion control mechanisms for their data packets: In an A<->B connection, data packets sent from A->B can use CCID 2, and data packets sent from B->A can use CCID 3.
- Different acknowledgement formats. The CCID for a connection determines how much ack information needs to be transmit-

ted. In CCID 2 (TCP-like), this is about one ack per 2 packets, and each ack must declare exactly which packets were received (Ack Vector option); in CCID 3 (TFRC), it's about one ack per RTT, and acks must declare at minimum just the lengths of recent loss intervals

- ★ Distinguishing different kinds of loss. A Data Dropped option lets one endpoint declare that a packet was dropped because of corruption, because of receive buffer overflow, and so on. This facilitates research into more appropriate rate-control responses for these non-network-congestion losses (although currently all losses will cause a congestion response)
- ★ Integrated support for mobility

Our testbed



IP: 192.168.1.xxx
Subnet Mask: 255.255.255.240

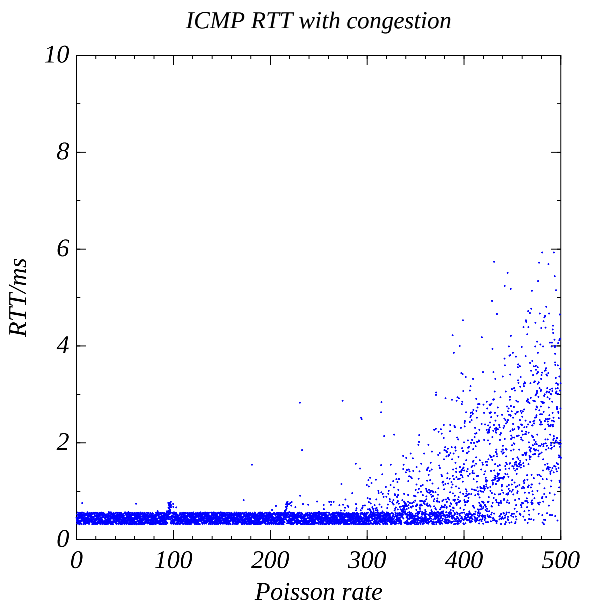
DCCP implementations

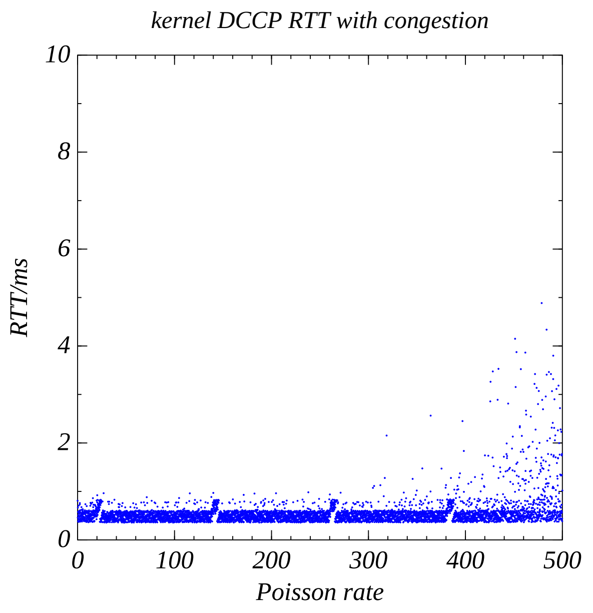
- McManus: kernel-space - requires patching the linux kernel
- Evlogimenis *et al.*: user space
- uses an underlying UDP socket
- single-threaded
- ask/choose cycle for option request (CCID)
- reliable acks a problem
  - pro: easy to debug
  - con: API differs from socket standard

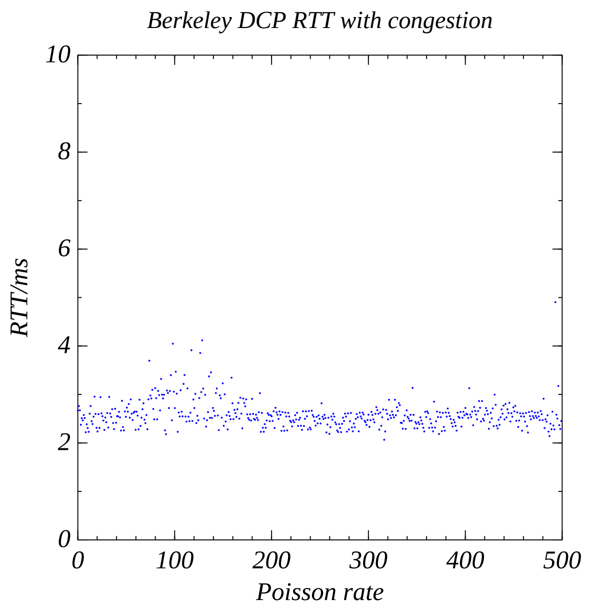
DCCP performance testing

- ★ We send traffic over several hops and measure RTT
- ★ We generate congestion traffic on intermediate links
- ★ Results below are for 4 hops, with Poisson traffic at λ 4kb packets
  per second

DCCP performance results







References

DCCP specs: http://www.icir.org/kohler/dccp/

Pedro Ruiz, Emilio Garcia: Improving user-perceived QoS in Mobile and wireless IP networking using real-time adaptive multimedia applications, 2002 http://www.agorasystems.com

D. Sisalem.: End-to-end quality of service control using adaptive applications, 1997

Schulzrinne *et al.*: RTP: A transport protocol for real-time applications, RFC 1889, 1996

Source

Authors: Timothy Sohn, Eiman Zolfaghari, Alkis Evlogimenos, Khian Hao Lim, Kevin Lai

`http://www.cs.berkeley.edu/~laik/projects/dccp/`

last updated: 2002 May 24

Version 0.0.1

Files required: `dcp.tar.gz`, `libkl-1.3.8.tar.gz`

blas and lapack for performance monitoring

Support documents: `http://www.icir.org/kohler/dcp/`

NB: build libkl first

Design

totally in user space - differs from McManus' implementation `http://www.ducksong.com:81/dccp/` which requires linux kernel patches

- pro: easy to debug
- con: API differs from socket standard

uses an underlying UDP socket

single-threaded

ask/choose cycle for option request (CCID)

reliable acks a problem

API - uses callbacks

1. **DCP_init** Initializes the library.
2. **DCP_main_loop** Enters the DCP_library main loop.
3. **DCP_socket** Creates a new socket.
4. **DCP_bind** Binds a socket to a port.
5. **DCP_listen** Sets up the socket for incoming connections.
6. **DCP_connect_handle** Sets up the success and error connect handlers for the specified socket.
7. **DCP_connect** Completes the connection of the specified socket.
8. **DCP_close_handle** Sets up the success and error close handlers for the specified socket.
9. **DCP_close** Closes the socket.

10. **DCP_accept_handle** Sets up the success and error accept handlers for the specified socket.

11. **DCP_accept** Creates a socket from an incoming request off the listening socket

12. **DCP_recv_handle** Sets up the success and error receive handlers for the specified socket.

13. **DCP_recv** Reads a packet's worth of data off the packet buffer.

14. **DCP_send_handle** Sets up success and error send handlers for the socket.

15. **DCP_send** Sends a packet of data.

16. **DCP_setsockopt_handle** Sets up success and error setsockopt handlers for the socket. It also specifies the changed option.

17. **DCP_getsockopt** Retrieves information about an option from the specified socket.

Instant Calls

DCP_init, *DCP_main_loop*, *DCP_socket*, *DCP_bind* and *DCP_listen* are instantaneous calls that do not require the use of callbacks.

DCP_init and *DCP_main_loop* functions are unrelated to any in the conventional socket interface. Given the user-level implementation, there is a need to initialize the underlying *UDP* socket and it is done through the *DCP_init* function. To suit the event-based engine, the application has to tie callbacks to events and finally enter a loop where all events take place. In our implementation, this loop happens as a last function call to *DCP_main_loop*.

Connect and Close

DCP_connect and *DCP_close* are slightly different from the rest. Their corresponding *handle* calls serve only to tie the application layer callbacks to the *DCP*-sockets. The *DCP_connect* call initiates the sending of the request packet and upon ending the proper starting sequence, would invoke the callback. The callback can then proceed to call the send and receive functions. The *DCP_close* call does essentially the same thing for the closing sequence.

Accept, Recv and Send

These three calls are similar in semantics. Their handlers also tie application level callbacks to the sockets but at the same time creates a holding back effect. This is best explained with an example. How the application could send a packet is by first calling *DCP_send_handle* which ties the sending callback to the socket. This is exactly the point where the *Congestion Control ID*'s control the flow of the packets. The *Congestion Control ID* in charge of sending only invokes the application's sending callback when sending is allowed. The application's sending callback would then call *DCP_send* which actually sends out the packet. *DCP_recv_handle* and *DCP_accept_handle* work similarly. They serve to register the desire of the application to receive or accept through the socket. When a packet arrives or when there is a new connection to accept, the application's callback would then be invoked. These would then call the corresponding calls of *DCP_recv* or *DCP_accept*.

getsockopt and setsockopt

DCP_setsockopt_handle is different from the rest of the calls. First, it does *not* have a corresponding *DCP_getsockopt* call. Requested options and callbacks are passed in the call to facilitate feature negotiation. The success or error callbacks serve to inform the application of the success or failure of the feature negotiation. The failure callback could be invoked upon receiving a CHOOSE and the application could then observe the CHOOSE values given and decide on a different feature value to negotiate. *DCP_getsockopt* then serves for the application to check the present feature values on the socket. Its semantics are very similar to that of the conventional socket interface.

Issues

Sequence numbers

- 24 bits $\Rightarrow$ 25GB
- 32 bits possible

User-space overheads

- buffer copying
- lack of timer resolution

the big question: should we write an interface so that we can easily use any application with this DCCP or the kernel version?

Example server code

dcp_simple_server0.c