Experimentation of the Datagram Control Protocol

Timothy Sohn, Eiman Zolfaghari
EECS Department
University of California at Berkeley
tsohn@cs.berkeley.edu, eiman@hkn.berkeley.edu

Abstract

This paper explores simulation results of the Datagram Control Protocol (DCP). The Internet community has seen the growth of real-time applications, which by the nature of their data flow do not desire the congestion control semantics of TCP and are thus left to build their applications on top of UDP. However, such methods have left with applications writers developing their own congestion control mechanisms. With congestion control mechanisms built directly into the transport layer, it provides ease of use for application writers, as well as a selection of standards for use by applications. We explore the advantages of DCP using an equation-based congestion control mechanism, TCP-friendly rate control. An initial implementation and experimentation of DCP and its equation-based congestion control mechanism shows its TCP-friendliness behaviors.

1. Introduction

Currently, a vast majority of the traffic contained in the Internet relies upon the Additive Increase-Multiplicative Decrease (AIMD) congestion control algorithm used in TCP, with UDP being the rest. However, due to the sharp rate changes which AIMD imposes, many of the applications which do not need an aggressive algorithm resort to using UDP and either implementing their own congestion control mechanism, or none at all. Moreover, in past years, the internet has seen the growth of real-time applications which are built upon UDP such as Counter Strike [22], Real Audio [23], and IP Telephony, just to name a few. If the greater majority of the traffic was UDP, this could present problems since 1) congestion control mechanisms are difficult to implement and 2) it is easy to have ill-behaved UDP flows which do not respond to congestion at all. However, since the volume of UDP traffic has been small relative to the rest of the internet's TCP flows, congestion collapse has not occurred.

Due to the fact that congestion control mechanisms are difficult to implement correctly, and application writers would like to avoid having to implement their own mechanisms for each of their applications, it would be useful to have the benefits of UDP, but with a selection of different congestion control mechanisms to choose from. Furthermore, to address the problem of ill-behaved UDP flows, a protocol which has incorporated within it, TCP-friendly flows would be beneficial to applications which require real-time streaming.

This problem has been addressed through DCP. This protocol was first proposed in [6] at ACIRI. However, due to the lack of an existing implementation or simulation based experiments, knowledge of how the protocol performs is unknown. DCP provides an unreliable flow of datagrams, with acknowledgements, to continue to accommodate current UDP applications, and also implements a congestion control manager. This congestion control manager provides the necessary interface for applications to choose what type of congestion control is desired for a flow. We refer to the different types of congestion control mechanisms as Congestion Control IDs (CCID). Currently two CCIDs have been proposed, CCID2, a TCP-like Congestion Control, and CCID3, a TCP-Friendly Rate Control (TFRC). The Congestion Manager is unique in that it allows multiple concurrent streams of data between the receiver and sender to share congestion mechanisms, or to have different congestion mechanisms. We note these as half-connections, signifying the bi-directional flow of data. Moreover, hosts are allowed to switch between different congestion control mechanisms on the fly, providing a type of modularity between the actual protocol and congestion control mechanisms.

We focus on the equation based congestion control mechanism, TFRC, which minimizes abrupt changes in the sending rate, while still maintaining long term fairness with TCP. We have implemented the DCP protocol along with the congestion manager and CCID3. Section 2 addresses the related work done in TCP-friendliness and equation based congestion control while Section 3 describes the design of DCP and our implementation. In Section 4, we describe our simulation

tests and experimentation results. Since there is still much more research to be done in DCP, Section 5 describes future work to be done in DCP concerning both implementation and modifications to the protocol. Finally we provide our conclusions in Section 6.

2. Related Work

Floyd et al. initially proposed and introduced the definition of TCP-friendly flows in [1]. Much of the work since then can be classified into proposals of transport protocols [12, 15, 16, 17], and congestion control mechanisms [7, 13].

Developments in the first area includes the Real-Time Transport Protocol (RTP) [15], which supports end-to-end delivery services for real-time data such as audio and video, and the Stream Control Transmission Protocol (SCTP) [16], which provides improved reliability using techniques such as multi-homing and accommodates multiple streams within a single end-to-end connection.

Applications typically would run RTP on top of UDP to make use of its checksum and multiplexing services. RTP itself does not provide any guarantees of timely delivery, nor provide other quality of service guarantees. Such a protocol would not be suitable for our needs to providing an alternative to the applications which currently use UDP, and supporting a standard way of implementing congestion control for applications, which would have otherwise had to implement their own congestion control (i.e. streaming media).

SCTP is an alternative to TCP, providing reliable, in-order delivery and multiple stream support. To that end, much of the inherent mechanisms in SCTP are unnecessary for an unreliable transport protocol. One idea is to modify SCTP's TCP-like semantics to allow out-of order delivery, and making changes to the congestion control mechanisms to support alternative mechanisms, such as a rate controlled mechanism. However, much of the variants of SCTP would still not make it suitable to flows currently using UDP, mainly because of efficiency reasons (SCTP overhead is large). Thus a new transport protocol, such as DCP would be needed to support unreliable delivery, and different built in congestion controls.

Research in the second area of congestion control mechanisms has centered on the TCP-Friendly Rate Control (TFRC) mechanism [7], where the sender adjusts its sending rate according to feedback of packet loss rates reported by the receiver over some period of time. Another mechanism is the Rate Adaptation Protocol [17], but unlike TFRC which relies on feedback packets sent over time, RAP implements an additive increase, multiplicative decrease algorithm. In addition, other mechanisms include a General Additive Increase, Multiplicative Decrase (GAIMD), which generalizes the TCP congestion window increase value and decrease ratio, and TCP Emulation at Receivers (TEAR) [13], which shifts flow control to the receiver and uses a sliding window to smooth sending rates. Simulations have been done comparing TFRC flows with TCP flows, through networks using RED and ECN [5, 11]. However, TFRC has not been used beyond simulation, and thus we validate the friendliness of TFRC (CCID3) on a real network with a network emulator, and show how DCP can be used with this CCID.

The current set of related work has either addressed building in a certain congestion control mechanism into the transport layer, or simply providing a congestion control mechanism for application designers to implement themselves. DCP attempts to bridge this gap by allowing the application writer to avoid implementing the congestion control, and instead simply picking the mechanisms he wishes to use directly from the transport protocol..

3. The Datagram Control Protocol

3.1 Protocol Overview

3.1.1 Main DCP ideas

The central idea to DCP is to allow applications to choose among several different forms of congestion control. Currently the two forms of congestion control supported are a TCP-like Additive Increase Multiplicative Decrease mechanism, and an equation based TFRC mechanism. In a DCP connection the CCID to use is negotiated upon, as well as other generic non CCID specific features of the protocol.

DCP has nine different packet types:

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

Mobility has been added to support host mobility and multihoming, although it is currently not in use.

Between the two communicating hosts there is an explicit connection setup and teardown. Furthermore, each side is able to establish its own half-connection, which consists of the data packets sent in one direction plus the corresponding acknowledgements coming back. For each half-connection, the client is able to then decide the congestion control mechanism to use (described as CCIDs). Specific connection properties are able to be negotiated, which are called features. DCP-Ask, Choose, and Answer are used in deciding upon what feature will be used for a DCP connection. Examples of features include negotiating a CCID, Timestamp, Ack Vector, or Mobility. Each side must agree to use the feature for it to be established.

3.1.2 CCID descriptions

Although there are currently two defined CCIDs, we focus on CCID3 – TFRC congestion control since this is what we have implemented.

CCID3 is meant to provide congestion control to those applications that send data packets to the end host without requiring a fully reliable connection, or that require a reliability mechanism on top of DCP. Ideally these applications would desire minute changes in the send rate exhibited by CCID3.

Once the CCID3 option has been negotiated by the sender and receiver, the connection would continue with the sender sending a stream of data packet to the receiver at an initial rate. The receiver sends a feedback packet to the sender at least once ever round-trip time. Using the information in the feedback packet, the sender adjusts its rate in accordance with the throughput equation given in [3]:

$$T = \frac{s}{R\sqrt{\frac{2bp}{3}} + t_{RTO}(3\sqrt{\frac{3bp}{8}})p(1+32p^2)} \qquad (1)$$

Where T is the send rate in bytes/second, s is the packet size in bytes, R is the round trip time in seconds, p is the loss event rate, t_{RTO} is the TCP retransmission timeout value in seconds, and b is the number of packets acknowledged by a single TCP acknowledgement, usually set to 1.

The receiver sends its receive rate estimate, and it's loss event rate which is then used by the sender. The sender keeps track of a no-feedback timer, and if no feedback packet is received within that time frame (usually two times the RTT), the send rate is halved.

Acknowledgements in TFRC are completely unreliable, and the data flow will continue regardless if acks are dropped or not.

3.1.3 Connection examples

We present an example of a typical DCP connection using CCID3.

1. The sender sends DCP-Data packets at a rate specified by its throughput equation. Enclosed in the packet is the sequence number and estimated round trip time value.

2. At least once every round trip time the receiver sends a DCP-ACK packet specifying its perceived receiving rate, the loss rate, and the sequence number of the last data packet received.

3. The sender continues to send packets and governs its transmit rate based upon the feedback packets. If no packets are received within two times the round trip time value, the send rate is halved.

3.2 Protocol Implementation

Our implementation of DCP uses an event-based model and is implemented in user space on top of UDP. We decided upon an event-based model rather than a thread-based one mainly for the ease of implementation. This does have some performance issues, with two memory copies required, one from the protocol to the library, then the library to the kernel. As a result of this choice though, the library is cleanly implemented; however, a certain drawback is for the application to adopt this same style of implementation in programming the application services. The interface to the congestion manager provides a layer of abstraction and modularizes the code to allow for further CCIDs to easily be built in.

Our implementation of CCID3 is based upon the TFRC code written in ns [20]. The original code written in ns is not designed to send real data, but instead just sends a blank packet of data when a timer goes off. One of the main difficulties we faced concerned timing and the accuracy of calculating round trip times. Although we use the RTC module for timing (which rounds to the nearest 10 ms), we found our timer callback did not exhibit the type of fine granularity TFRC required. Therefore, in our implementation, while the sender is waiting to send the next packet, it may wait beyond the required time. Thus it was necessary to calculate the time the sender actually waited before sending a packet and if delayed such that the sender is sending lower than the perceived rate, we would exhibit a packet burst to match the expected rate.

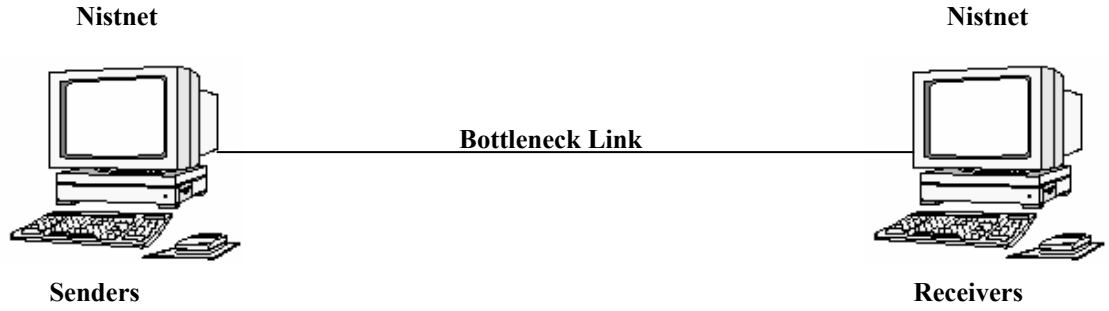


Figure 1: Network Topology

Another issue we faced involved the calculation of the round trip time. When the sender enqueues a packet into the underlying library to be sent, there is no guarantee the packet will be sent right away. Furthermore, since the sender starts the round trip timer immediately when enqueuing the packet to be sent, there is a certain amount of inaccuracy involved. However, we found that our round trip time estimates are only slightly greater than what is actually expected

When comparing our results to those given in simulation by ns, we found that the round trip time of ns stabilizes rather quickly, and provides smoother results. Although our results differ due to slightly higher and constant minute changes in the round trip time, we found our results to verify the friendliness, compared to other previous related work, of a rate-based scheme in the real-world setting.

4. Emulation and Experimental Results

In this section we present simulation and experimental results of the DCP protocol. We performed these simulations to study the TCP-friendly behavior of our protocol in a small environment.

4.1 Network Topology

Our measurements were performed with two machines running Linux, connected by a single 100 Mb Ethernet link.

As opposed to choosing a simulator to test our implementation, which involves a totally synthetic environment, we used the Nistnet emulator [21] to control our experiments, setting the delay, bandwidth, and queue size, imitating the performance characteristics of real networks. As opposed to a drop-tail or RED mechanism for its queues, Nistnet uses a Derivative Random Drop (DRD) style mechanism. Two values are defined, the DRDmin and DRDmax. No packets are dropped if the queue length is below the minimum value and 95% of packets are dropped if above the max value. The drop percentage ramps up linearly as the queue length increases between these two values. Due to Nistnet's behavior of only shaping incoming packets, we had to run the emulator on both machines to accurately emulate the network.

During our simulation studies, we encountered issues with Nistnet when sending multiple TCP flows. Therefore, we were unfortunately unable to expand to a greater number of flows, but we believe our scenarios are evident enough to display the friendliness characteristics of DCP.

To extract information while running our simulations, we used *tcpdump* [24] and perl scripts. Tcpdump simply dumped all the packets that a certain network interface saw. For each scenario we tested, we ran tcpdump in the background and saved the dump to a file. The perl scripts parsed through this file and, depending on the settings given to it, created files which, when given to *gnuplot*, graphed a certain metric.

4.2 Performance metrics

Our measurement of throughput is given by previous characterization of TCP-friendly protocol throughputs in [18]. This equation characterizes the steady-state throughput of a TCP-connection in an environment without timeouts as:

$$Throughput = \frac{C}{R \times \sqrt{p}} \qquad (2)$$

where C is a constant set to 1.22, R is the round trip time for the connection, and p is the packet loss rate. Since the equation assumes timeouts do not occur, the equation is

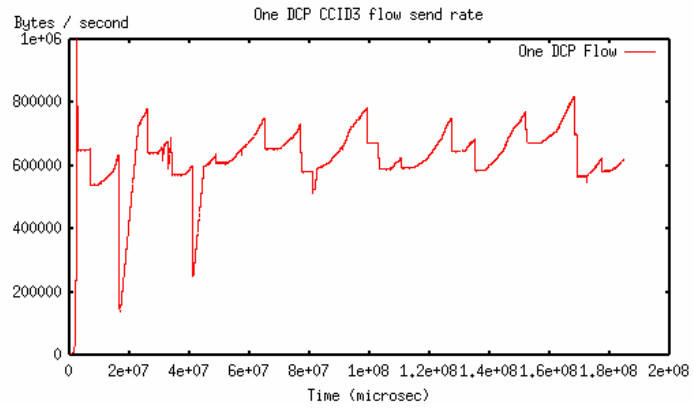
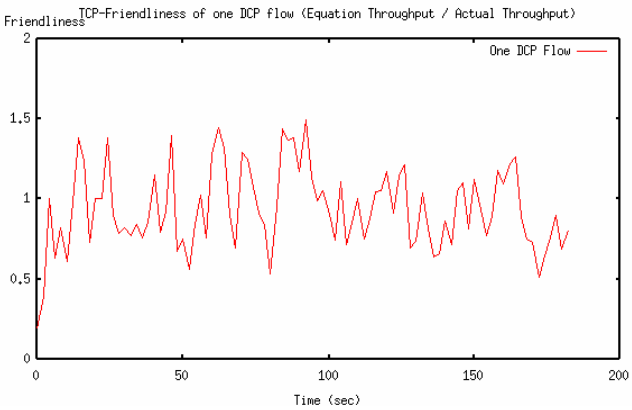


Figure 2 : One DCP CCID3 flow send rate



Average TCP-Friendliness of one DCP flow = 0.9374

Figure 3: TCP-Friendliness of one DCP CCID3 flow

somewhat inaccurate in estimating throughput above a loss rate of 5% as reported in [19].

Our measure for analyzing friendliness is based upon those provided in [2]. We define the friendliness ratio of CCID3 without the presence of background flows to be:

$$F = \frac{T_{calculated}}{T_{actual}} \qquad (3)$$

Where $T_{calculated}$ is the DCP CCID3 throughput calculated in (2), and T_{actual} is the measured as packets per second. This is done through using a six second window and measuring the number of packets sent during that window of time.

In the presence of other TCP flows, we define the following friendliness metrics. For each DCP connection we denote the throughput of each flow as $T_1^D, T_2^D, \ldots, T_k^D$ where k is the number of connection, and similarly for TCP, $T_1^T, T_2^T, \ldots, T_n^T$ where n is the number of TCP connections. The respective throughput then of the DCP and TCP flows is

$$T_D = \frac{\sum_{i=1}^{k} T_i^D}{k} \quad \text{and} \quad T_D = \frac{\sum_{i=1}^{k} T_i^D}{k} \qquad (4)$$

Friendliness between DCP and TCP flows is then defined to be:

$$F = \frac{T_D}{T_T} \qquad (5)$$

4.3 Emulation Scenarios

Our testing scenarios were meant to validate the TCP-friendly results of previous work, and expand upon how DCP behaves with CCID3. We aimed to look at three sets of tests: one DCP flow alone, one DCP flow with N TCP flows, and application end-to-end delay measurements of DCP in comparison to other protocols (UDP, TCP). All our measurements were done through Nistnet emulating a 50 millisecond delay, 10Mb link, and a queue size roughly as large as the bandwidth delay product. Our reason for using these parameters were the current average connection of an end user in the Internet does not extend beyond 10Mbps, let alone 1.5Mbps on a broadband (DSL, cable) connection. Furthermore, we generally see 50ms delays across the internet, for a "good" connection. We hope to at one point test the protocol over higher bandwidth links with lower delays.

Our tests are described more thoroughly below:

One DCP flow – We measured a 200 second length DCP flow by itself without any background flows. Our purpose for this test was to analyze the throughput of the protocol in terms of stability and bandwidth utilization.

One DCP flow with N TCP flow(s) – Here again we measured a 200 second long DCP flow, but now in the presence of various number of TCP flows. We varied this number from 1-9. Although we would have liked to increase this number, we found many complications out of our control due to Nistnet.

Delay - We anticipated examining end to end delay overhead involved with DCP and comparing that to UDP and TCP. However, due to time constraints, we were unable to do such a test, but have described it here in detail

for the reader. Our first attempt to measure overhead involved simply sending a 1500 byte packet to the receiver, and having the receiver send another 1500 byte packet in return, and then measuring the total time it took for the applications to send and receive these two messages. We found however, this methodology was unfair in some sense because 1) DCP is implemented in user space, incurring some greater amount of delay than the other two protocols in the kernel, and 2) something… We feel a slightly fairer test would be to force a total send of 15KB and measuring the end to end delay for every 10th packet (which would be 1500 bytes). We feel this is fairer since it takes advantage of TCP's retransmission mechanism, and CCID3's congestion control. We anticipate the results to show UDP with the least overhead, then DCP, and finally TCP, mainly due to TCP's retransmissions.

Throughout all of our tests, TCP sends packet sizes of 1500 bytes and DCP at 1000 bytes.

4.4 Results

The scenarios we tested have shed light on our implementation of DCP CCID3 and how it works with one DCP flow and along with multiple competing TCP flows.

Figure 2 provides the sender's estimate of its send rate contained within each packet. In the beginning, the sender estimates too large of a rate due to the throughput equation in (1) which is calculated with an RTT and loss value of 0. However, CCID3 eventually begins to converge around 600KB over time. We anticipated DCP to be able to use the full bandwidth link, however, due to the nature of the equation governing the send rate, it seems reasonable for the flow to have 60-80% bandwidth utilization. Unlike previous work, our implementation tended to respond quite quickly to congestion in the link, mainly due to overflowing of the queue. However, this quick response, provides a more accurate send rate, without having to go through the sharp, large changed involved in an AIMD mechanism.

This sort of hovering over a certain value is evident in Figure 3. Here we have taken the equation throughput of (2) and used it to estimate the value which TFRC should achieve in order to have similar throughput to TCP. We then calculated the average actual throughput of DCP in six second intervals. Our desire is to have a friendliness value of 1, which we do see as the flow fluctuates around that value. The average friendliness across the entire session is approximately 94%. Therefore, DCP maintains a relative friendliness

Oddly, the amount of time taken for DCP to stabilize to some approximate rate took quite a long time, and from analyzing Figure 2, it could be said that the flow took the entire 3 minutes of the connection. Judging from both Figure 2 and Figure 3, it seems that CCID3 needs a great length of time to converge to a value. This of course is unfortunate for video streaming applications that may only stream a clip for only 30 seconds or less.

When TCP flows are introduced, CCID3 continues to act TCP-friendly. We see in Figure 4 that as the number of TCP connections increases, DCP CCID3 lowers its sendrate and allows the other TCP flows to take more bandwidth. The graphs plot the aggregate of all TCP flows verses the single DCP flow running. Our results are quite promising of the potential for CCID3 to meet the needs of current applications in need of an equation based congestion control built into the transport protocol. The first graph shows DCP competing equally with TCP for a fair share of the bandwidth. As the number of TCP flows begins the increase, the amount of bandwidth DCP uses begins to decrease.

Convergence seems to be a bit faster for the protocol in the presence of many more background flows. As the number of TCP flows increases, DCP converges faster towards an estimated fair rate. This can be seen from the graphs of a one DCP/one TCP connection to a one DCP/nine TCP connection. Furthermore, looking at figure 5, we see evidence of friendliness smoothing out to a value of 1. In the presence of just one TCP flow, DCP tends to be slightly more friendly than it should be, but able to still send at a competitive rate.

In all of our results, we found CCID3 able to maintain a fair rate, and to compete well with TCP connections. Hence the issues of accommodating real-time applications to have a TCP-friendly rate, yet still be competitive with other background connections is done well with DCP. One can imagine the broad range of congestion control mechanisms that could be built into DCP and analyzing the friendliness equation in the presence of different CCIDs. Furthermore, we have provided an easy way for current UDP applications to simply transfer over to DCP and simply specify a desirable congestion mechanism.

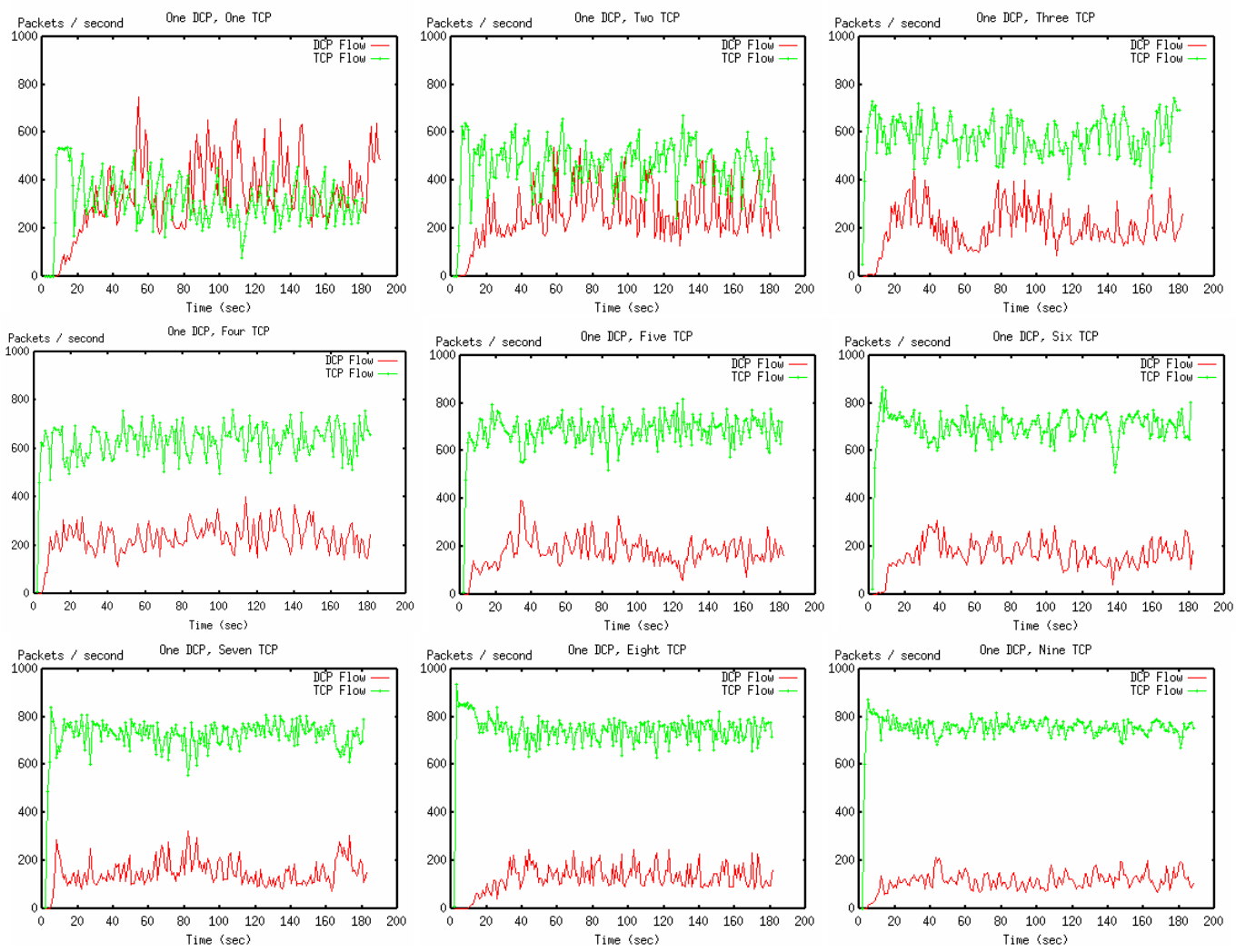


Figure 4: One DCP CCID3 Flow in Competition with many (1-9) TCP flows

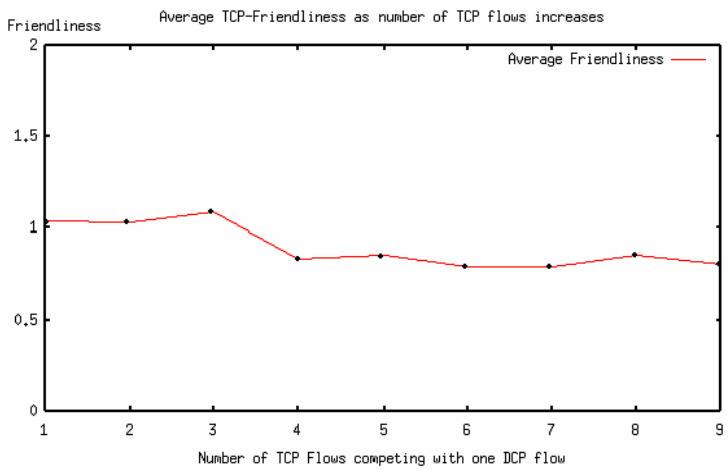


Figure 5: Average TCP-Friendliness as the number of TCP flows increases

5. Future Work

There are many areas for the DCP implementation to be expanded. We are currently working to complete the implementation of CCID2, TCP-like congestion control With the completion of CCID2 it is possible to explore the benefits and use of half-connections, to mix CCIDs between the sender and receiver. As an example, with an asymmetric connection, if the receiver can receive at a high bandwidth, but can only send back acknowledgements at a low rate, the receiver could implore a CCID2 mechanism for use on one link, while using a CCID3 mechanism on the other. Moreover, additional rate-based CCIDs could be added to the protocol, such as GAIMD. DCP provides this for of scalability for applications to select from a wide variety of congestion control mechanisms.

Other features of DCP which have yet to be explored pertain to acknowledgements. Currently DCP contains an Ack Vector feature, which allows the receiver to provide the sender with detailed loss information and lets senders quickly report back to their senders when packets are dropped. The Ack ratio option allows two sides to negotiate the frequency of acknowledgements for data packs.

Since our current implementation of DCP is placed in user-space, it incurs a great deal of context switching between the protocol and the operating system kernel. We hope to move our implementation into the kernel and analyze performance results of such a change.

Finally, because DCP is directed towards those applications which currently use UDP without any form of end-to-end congestion control, an area of interest would be implementing a layer of reliability on top of the DCP layer.

6. Conclusion

DCP provides applications which currently use UDP with a transport layer protocol that allows its own specification of various congestion control protocols. We have provided results of just one of the many CCIDs to be proposed, and have shown how an application would use this protocol. CCID3 achieves friendliness through a rate based adjustment equation using the loss rate and round trip time estimate sent by the receiver. In the previous section we outlined future work which would further display the various features in DCP, and provide a better incentive to current applications to switch to this transport protocol.

7. Acknowledgements

We would like to acknowledge Kevin Lai for his guidance and direction throughout this project. We also thank Alkis Evlogimenos and Khianhao Lim for their help in implementing the base of the DCP protocol. Thanks also to Stephan Baucke of Ericsson, for his suggestions in our work.

8. References

[1] S. Floyd and K. Fall, "*Promoting the Use of End-to-End Congestion Control in the Internet*," IEEE/ACM Transactions on Networking, August 1999.

[2] J. Padhye, J. Kurose, D. Towsley, and R. Koodli. *A model based TCP-friendly rate control protocol*. In Proc. International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV), Basking Ridge, NJ, June 1999.

[3] S. Floyd, M. Handley, J. Padhye, and J. Widmer. *Equation based Congestion Control for Unicast Applications*. In Proceedings of ACM SIGCOMM, Stockholm, Sweden, August, 2000.

[4] K. Fall and S. Floyd. *Simulation-based comparisons of Tahoe, Reno and SACK TCP*. Computer Communications Review, vol. 26, no. 3, pp. 5-21, July 1996.

[5] S. Floyd, M. Handley, and J. Padhye. *A Comparison of Equation-Based and AIMD CongestionControl*. http://www.aciri.org/tfrc/, May 2000.

[6] S. Floyd, M. Handley, and E. Kohler. *Problem Statement for DCP*. http://www.icir.org/kohler/dcp, February 2002

[7] M. Handley, J. Padhye, and S. Floyd. *TCP Friendly Rate Control (TFRC): Protocol Specification*. http://www.aciri.org/tfrc/, July 2001

[8] E. Kohler, M. Handley, S. Floyd, and J. Padhye. *Datagram Control Protocol*. http://www.icir.org/kohler/dcp, November 2001

[9] J. Padhye, S. Floyd, and E. Kohler. *Profile for Congestion Control ID 3: TFRC Congestion Control*. http://www.icir.org/kohler/dcp/, March 2002

[10] D. Sisalem and H. Schulzrinne. *The loss-delay based adjustment algorithm: A TCP-friendly adaptation scheme*. Workshop on Network and Operating System Support for Digital Audio and Video, 1998.

[11] S. Hassan and M. Kara. *Simulation-based Performance Comparison of TCP-Friendly Congestion Control Protocols*. In the Proc. of the 16th Annual UK Performance Engineering Workshop (UKPEW2000), Durham, UK, July 2000.

[12] G. Wong, M. Hiltunen, and R. Schlichting. *A configurable and extensible transport protocol*. In IEEE Infocom 2001, Anchorage, Alaska, April 2001.

[13] I. Rhee, V. Ozdemir, and Y. Yi. *TEAR: TCP Emulation at Receivers -- Flow Control for Multimedia Streaming*, NCSU Technical Report, April 2000

[14] Y. R. Yang, M. S. Kim, and S. S. Lam. *Transient behavior of TCP-friendly congestion control protocols*. In Proceedings of IEEE INFOCOM, April 2001.

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

[16] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I Rytina, M. Kalla, L. Zhang, and V. Paxson. *Stream Control Transmission Protocol*. RFC2960

[17] R. Rejaie, M. Handley, and D. Estrin. *RAP: An End-to-end Rate-based Congestion Control Mechanism for Realtime Streams in the Internet*. In Proceedings of IEEE INFOCOMM, New York City, NY, March 1999.

[18] M. Mathis, J.Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *Computer Communication Review,* 27(3), July 1997.

[19] J. Mahdavi and S. Floyd. TCP-friendly unicast rate-based flow control. Note sent to end2end-instrest mailing list, Jan 1997

[20] *ns-2 simulator* http://www.isi.edu/nsnam/ns

[21] *NIST Net* http://snad.ncsl.nist.gov/itg/nistnet

[22] *RealAudio Player* http://www.real.com

[23] *Counter Strike* http://www.counter-strike.net

[24] *tcpdump* http://www.tcpdump.org