Making large scale deployment of RCP practical for real networks

Chia-Hui Tai, Jiang Zhu, Nandita Dukkipati
Computer Systems Laboratory
Department of Electrical Engineering
Stanford University
{chtai, jiangzhu, nanditad}@stanford.edu

***Abstract*—We recently proposed the Rate Control Protocol (RCP) as a way to minimize download times (or flow-completion times). Simulations suggest that if RCP were widely deployed, downloads would frequently finish an order of magnitude faster than with TCP. This is because RCP involves explicit feedback from the routers along the path, allowing a sender to pick a fast starting rate, and adapt quickly to network conditions. RCP is particularly appealing because it can be shown to be stable under broad operating conditions, and its performance is independent of the flow-size distribution and the RTT. Although it requires changes to the routers, the changes are small: The routers keep no per-flow state or per-flow queues, and the per-packet processing is minimal.**

However, the bar is high for a new congestion control mechanism – introducing a new scheme requires enormous change, and the argument needs to be compelling. And so, to enable incremental deployment of RCP, we have built and tested an open and public implementation of RCP, and proposed solutions for deployments that require no fork-lift network upgrades.

In this paper we describe our end-host and router implementation of RCP in Linux, and solutions to how RCP can coexist in a network carrying predominantly non-RCP traffic, and coordinate with routers that don't implement RCP. We hope that these solutions will take us closer to having an impact in real networks, not just for RCP but also for many other explicit congestion control protocols proposed in literature.

I. Introduction and Motivation

We want to enable deployment of RCP (Rate Control Protocol) congestion control in real networks. As with any new congestion control mechanism, there are a slew of questions to answer: What are its benefits over existing TCP mechanisms? Is it stable under sudden network changes? Does it scale with network bandwidth-delay product? How complex is it to implement in real end-hosts and routers? Can it be incrementally deployed in real networks?

In prior work, we studied RCP through simulations and modeling; in particular scalability and stability of RCP have been proved, and thousands of simulations suggest it is very promising under a broad range of conditions. In this work we are interested in the practical considerations of an Internet-scale deployment of RCP: how complex is it to implement RCP, and how can it be incrementally deployed in real networks.

Before explaining how to make RCP practical, we need to understand what RCP is and how it works.

In the basic RCP algorithm a router maintains a single rate, $R(t)$, for every link. The router "stamps" $R(t)$ on every passing packet (unless it already carries a slower value). The receiver sends the value back to the sender so that it knows the slowest (or bottleneck) rate along the path. In this way, the sender quickly finds out the rate it should be using (without the need for slow-start). The router updates $R(t)$ approximately once per roundtrip time (RTT), so as to emulate processor sharing among flows. Intuitively, to emulate processor sharing, the router should offer the same rate to every flow, try to fill the outgoing link with traffic, and keep the queue occupancy close to zero. The RCP rate update equation is based on this intuition:

$$R(t) = R(t-T)(1 + \frac{\frac{T}{d}(\alpha \cdot (\gamma \cdot C - y(t)) - \beta \cdot \frac{q(t)}{d})}{\gamma \cdot C}) \quad (1)$$

where d is a moving average of the RTT measured across all packets (each RCP sender maintains its RTT estimate which it stamps in all outgoing data packets), T is the update interval (i.e., how often $R(t)$ is updated) and is less than or equals d, $R(t-T)$ is the last rate, C is the link-capacity, $y(t)$ is the measured aggregate input traffic rate during the last update interval, $q(t)$ is the instantaneous queue size, α, β are parameters chosen for stability and performance, and γ controls the peak link-utilization.

There are four main features of RCP that make it an appealing and practical congestion control algorithm:

1) RCP is inherently fair (all flows at a bottleneck receive the same rate).
2) RCP's flow-completion times are often one to two orders of magnitude better than in TCP-Sack and XCP [7], and close to what flows would have achieved if they were ideally processor shared. This is because RCP allows flows to jump-start to the correct rate (because even connection set-up packets are stamped with the fair-share rate). Even short-lived flows that perform badly under TCP (because they never leave slow-start) will finish quickly with RCP. And equally importantly, RCP allows flows to adapt quickly to dynamic network conditions in that it quickly grabs spare capacity when available and backs off by the right amount when there

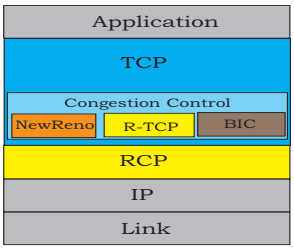

Fig. 1. RCP is a protocol between the IP and transport layers.

is congestion, so flows don't waste RTTs in figuring out their transmission rate.

3) There is no per-flow state or per-flow queueing.
4) The per-packet computations at RCP router are simple.

Although we have many thousands of promising $ns2$ [13] simulations, we want to find out how feasible it is to deploy RCP in real networks. Deploying RCP involves solving many practical problems. To that end, the focus of this paper is on two problems, both key to achieving our goal: How is RCP implemented in real systems and what is the additional complexity it introduces in end-hosts and routers (Section II)? How does RCP coexist in a network where a significant portion of traffic is non-RCP and a large portion of queues do not yet implement RCP (Section III)?

II. Implementing and Experimenting with RCP

We recently described a linux based implementation of RCP end-host and router [5], and a hardware implementation of RCP router on Stanford's NetFPGA system [12]. Our goal here is to describe the testbed we are building with these implementations, along with experiments we are conducting to both validate the implementation as well as to verify that RCP performs as we would expect it to. We will first give a brief description of our implementation and analyze the additional complexity it introduces into end-host software and routers.

A. Implementing RCP end-host and router

1) RCP End-host: We have implemented the RCP end-system in Linux 2.6.16. An RCP sender maintains a congestion-window which it modulates based on explicit feedback information from the network. It also maintains a round-trip time estimate of the path and paces a window's worth of packets within a RTT. An RCP receiver echoes the network rate feedback it receives to the sender by piggybacking it in the reverse DATA/ACK packets. We describe below the key pieces of an RCP end-system.

RCP is implemented as its own protocol layer between IP and transport layers, as shown in Figure 1. Other places to carry RCP information would be IP or TCP options, each having its pros and cons.[1]

[1]The advantages of having RCP as a shim layer between IP and transport are: a) routers which don't understand RCP will let RCP packets pass through, and b) the RCP rate information can be used by any transport protocol including TCP for file-transfers, as well as by UDP for streaming content. Figure 2 illustrates the 12-Byte RCP congestion header, having four fields.

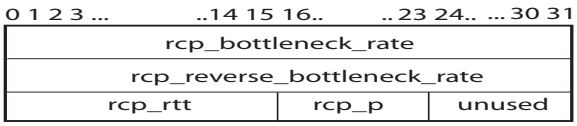

Fig. 2. The 12-Byte RCP header: the *rcp_bottleneck_rate* (4 Bytes) carries the rate (in Bytes/msec) of the most congested link along the path; *rcp_reverse_bottleneck_rate* (4 Bytes) is the bottleneck rate (in Bytes/msec) echoed by the receiver, so the sender can adapt its rate; *rcp_rtt* (2 Bytes) is the sender's estimate of its round-trip time (in msecs); *rcp_proto* (1 Byte) is the protocol number of the higher transport layer.

When describing the RCP implementation at the end-host, we will focus here on the case of TCP transport protocol running on top of RCP. Figure 1 shows the placement of RCP in the network stack; there are two parts: the RCP layer between transport and IP layer, which carries congestion information from network to the end-system, and the congestion control component in transport layer which adapts the flow-rate based on network feedback. One can think of congestion control consisting of two broad parts: a) modulating the flow-rate (and congestion window), and b) deciding which packets to send among the three pools of packets - those which have not yet been transmitted, those which have been sent but not yet acknowledged, and finally packets which are known to be lost. RCP only modifies the first of these functions in TCP, i.e. modulating the flow-rate, and we call this part as *R-TCP*. Starting from Linux 2.6.13, the TCP code was re-written to make it more modular [8], as a result of which the specific TCP congestion control mechanism, for e.g. BicTCP, HTCP, Scalable TCP, HighSpeed TCP, can be chosen dynamically either using $sysctl$ or on a per-socket basis. *R-TCP* can also be chosen dynamically. The rest of TCP functionality such as the state-machine and mechanisms for in-order packet delivery remain unchanged.

An RCP sender maintains the following variables: a) bottleneck rate of the forward path, b) bottleneck rate of the reverse path, c) round-trip time estimate for the current path, and d) the packet pacing interval. These are maintained in TCP's tcp_sock structure. The sender fills in RCP fields of an outgoing packet: a) sender's desired throughput, $rcp\_bottleneck\_rate$, which can be the speed of the local interface, b) bottleneck rate of the reverse path, $rcp\_reverse\_bottleneck\_rate$, which is zero if the host is not aware of the rate yet, c) round-trip time estimate, which is zero for the first packet of the connection (SYN) when the sender does not have an estimate yet, and d) the protocol number of TCP. RCP intercepts all calls from TCP to IP to attach the 12-Byte RCP header, and similarly detaches the RCP header before passing on packets to TCP from IP.

An RCP receiver echoes the network rate feedback to the sender by copying the $rcp\_bottleneck\_rate$ value into $rcp\_reverse\_bottleneck\_rate$, and usually piggybacking on DATA/ACK packets. For a pure ACK packet, the bottleneck rate and RTT fields are set to zero.

On receiving valid rate feedback, *R-TCP* modulates its

congestion window,[2] overriding the slow-start and congestion avoidance window changes. *R-TCP* keeps track of the smoothed round-trip time estimate for this connection, and paces packets from TCP's send queue. The mechanisms that decide which packets to retransmit upon losses remain the same as in TCP.

2) RCP Software Router: This section outlines the implementation of an RCP router in software, based on the specification [2] [6]. Such an implementation demonstrates the feasibility and simplicity of supporting RCP within a software-based router.

Besides the normal operations of a linux-based IP router, such as the longest prefix match look-up, and forwarding packets to the destination interface, an RCP router must additionally compute the fair-share rate periodically (as per Equation 1) and stamp that rate in packet headers. The rate is computed once every control interval which is in the order of a round-trip time. During this control interval the router collects a few statistics, average RTT of traffic and total incoming bytes destined to an output port, which it then uses in computing the rate.

We run our RCP router implementation on a standard Linux system, implemented as a Linux Kernel Module (LKM), namely *rcp-router-driver.ko*. This approach avoids the complication of applying patches to Linux source distribution and recompiling the whole Linux kernel. The RCP implementation consists of two main parts:

1) Per-packet handling in the data-path which includes:
   - Identifying whether an incoming packet is an RCP packet
   - Updating a running RTT sum of the outgoing interface if the packet carries a valid RTT
   - Updating the aggregate traffic destined to the outgoing interface
   - Stamping the RCP rate in the outgoing packet
2) Periodic computations in the control path: The router periodically (approximately once per average RTT of sessions passing through it, but more frequently if appropriate) calculates how much bandwidth it can allocate, $R(t)$, to the average data flow, as per Equation 1. Periodic computations also include updating the moving round-trip time average, as detailed in [6].

The control plane is a timer driven function to compute the RCP rate, moving RTT average on each outgoing interface, and the next wake-up interval for this timer. The timer is maintained per network interface. The Data plane is built based on Linux's NetFilter feature, which allows customized per-packet operations in packet processing chain of the kernel. RCP requires only a small amount of per-packet processing – in the worst case 3 integer additions, 2 comparisons, and 1 write operation. No multiplications or divisions are performed on the data path. Data plane operations on Ingress and Egress path are described below:

[2]snd_wnd = (rcp_bottleneck_rate × rcp_rtt) / (MSS + RCP_HEADER_SIZE + IP_HEADER_SIZE)

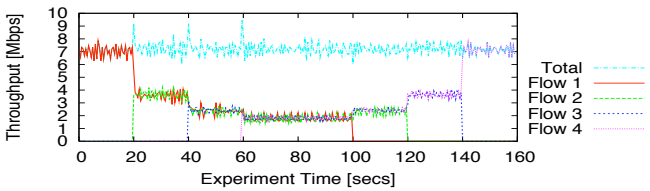

Fig. 3. Multiple end-hosts are sending traffic to a sink through an RCP router. The router is configured to a bottleneck link rate of $C = 7.5$Mbps. End-hosts start flows of 100s duration at times 0s, 20s, 40s and 60s respectively. The plot shows the per-flow throughput and aggregate throughput over time as flows arrive and depart at the RCP router. The arrival of new flows create sudden backlogs at the router's TX queue, in response to which RCP reduces $R(t)$ to approximate C/N.

- The Ingress function is registered with *IP_FORWARDING* hook in NetFilter. When a packet arrives at the NIC driver and is destined to one of the outgoing interfaces, this function updates running RTT sum of the outgoing interface (if the packet carries valid RTT); it also updates the aggregate traffic rate to the outgoing interface.
- The Egress function is registered with *IP_POSTROUTING* hook in NetFilter. When a packet is ready for one of the outgoing interfaces, this function stamps RCP rate in the header and updates the TX queue occupancy for that interface.

3) Complexity of RCP End-host and RCP software router: The RCP end-host has 250 lines of C code (including commenting and declarations), and does not involve any floating point computations. For the RCP router, the computations on the control plane (for periodic RCP rate and average RTT calculations) and the data plane for each transit packet through the router are shown in Table I.

TABLE I
COMPLEXITY OF RCP SOFTWARE ROUTER

	Control Plane	Data Plane		
		Ingress	Egress	Total
LOC	28	11	13	24
U32 Comparison	3	3	2	5
U32 Additions	5	4	0	4
U32 Multiplication	26	0	0	0
U32 Assignments	9	5	6	11

To compare the complexities between a standard linux router without RCP support and an RCP router: for a non RCP-enabled kernel, each packet processing in the IP forwarding path takes 9.7368 jiffies, and with RCP it takes 9.9998 jiffies. Therefore, RCP-related processing is only 2.6% of the IP packet forwarding in the kernel.

B. Example Experiment

Figure 3 illustrates the results of an example RCP experiment on our test-bed. Other experiments on a range of topologies, heterogeneous round-trip times and different synthetic traffic patterns to verify both RCP's strengths (short flow-completion times) and weaknesses (transient queues on sudden changes) are in [10].

III. INCREMENTALLY DEPLOYING RCP

For RCP to receive widespread adoption it would need to be introduced into a network with two hindrances: (1) RCP will

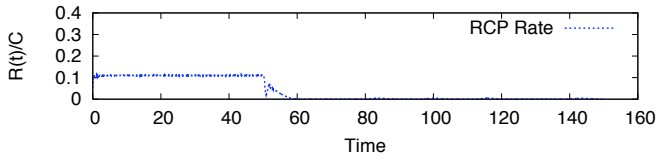

Fig. 4. Demonstrating the problem when RCP and TCP coexist: Bottleneck link implements the RCP algorithm. Nine RCP flows start at time $t = 0$, and one TCP flow joins in at $t = 50$. RCP rate is throttled immediately.

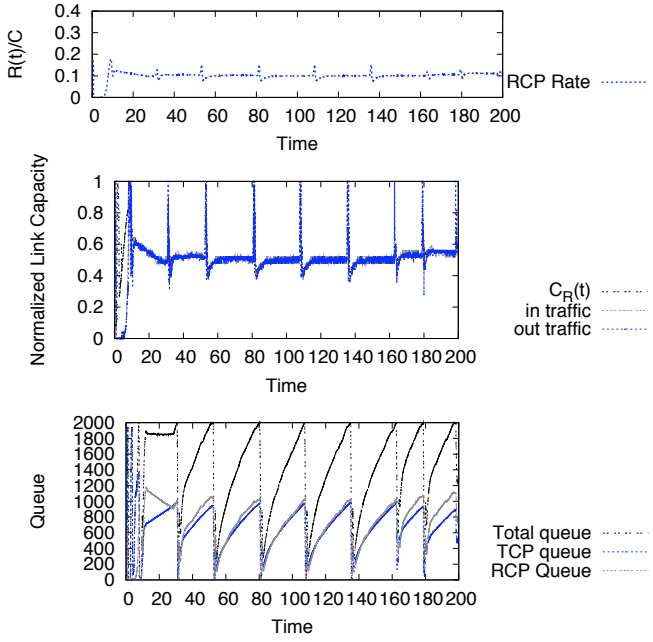

Fig. 5. Example showing the deployment solution in action: In a single-hop network are five RCP long-lived flows and five TCP long-lived flows. The average link shares are 0.5 each for RCP and TCP. The fluctuations follow from TCP's saw-tooth behavior.

need to operate alongside existing non-RCP traffic, such as TCP and UDP, without adversely affecting or being affected by the other traffic; and (2) RCP will need to operate in a network where some routers are not RCP-enabled. In this section, we will explore both hindrances in turn.

A. Hindrance 1: RCP must coexist with non-RCP traffic

Let's first understand how severe the hindrance can be. Imagine a simple network in which all routers are RCP-enabled, but must carry both RCP and TCP traffic. Figure 4 illustrates an example, where nine long-lived RCP flows share a link from time 0, and a TCP flow joins the network at time 50. The RCP flows are throttled. This is because TCP fills up router buffers until they overflow, while RCP attempts to keep the buffer occupancy low and only makes use of the spare capacity. On seeing queued-up TCP packets and increasing incoming traffic, RCP backs off and eventually stifles its own transmission rate.

We have explored a wide range of solutions to this problem and propose a few modifications to RCP routers.

The objective of our solution is for RCP to achieve a fair link share – regardless of how other types of traffic behave, and without requiring to isolate the different kinds of traffic. For this purpose, RCP must explicitly know the proportion of link-rate available to it so as to achieve the same per-flow rate for RCP and non-RCP traffic. The link-rate available to RCP, denoted as C_R, has to be updated frequently with varying traffic conditions, and be used in RCP's rate equation (in place of C). We formalize this intuition below:

- $y_R(t)$ denotes the measured incoming RCP traffic-rate, $C_R(t)$ is the link-rate available to RCP, and $q'_R(t)$ is the RCP virtual queue length in terms of packets.
- **STEP C**: Once every control period T_c, do the following:
  (C.1) Estimate the approximate number of active RCP and TCP flows, denoted as $N_R(t)$ and $N_T(t)$.
  (C.2) Find the weight, w_R, to adaptively equalize the average flow rate of RCP ($r_R(t)$) and TCP ($r_T(t)$).

$$r_R(t) = \frac{y_R(t)}{N_R(t)}$$
$$r_T(t) = \frac{y_T(t)}{N_T(t)}$$
$$w_R = w_R - \theta \frac{r_R(t) - r_T(t)}{r_R(t) + r_T(t)}$$

  where θ is a parameter defined in $(0, 1)$, controlling how smooth the update is.
  (C.3) Update C_R as follows:

$$C_R(t) = \max(C - y_T(t), C \cdot w_R)$$

- **STEP Q**: Evolve $q'_R(t)$ upon the n-th RCP packet arrival at time t_n,

$$q'_R(t_n) = [q'_R(t_{n-1}) - C_R(t) \cdot (t_n - t_{n-1})]^+ + 1$$

- **STEP R**: Once every rate-update interval, T, compute the RCP rate:

$$R(t) = R(t-T)\left(1 + \frac{\frac{T}{d_0}[\alpha(\gamma \cdot C_R(t) - y_R(t)) - \beta \frac{q'_R(t)}{d_0}]}{\gamma \cdot C_R(t)}\right)$$

In Step (C.1), we estimate the number of flows $N_R(t)$ and $N_C(t)$ using a randomized algorithm with Zombie list proposed in [9]. A zombie list is a fixed size array recording the flow identifiers of recently seen packets. On each packet arrival, the flow id either writes to an empty entry or, with probability p, overwrites a stored entry if the list is full. The number of flows, $N_R(t)$ and $N_T(t)$, is estimated by the reciprocal of the hit probability, the probability that an incoming packet belongs to a flow in the list. Other algorithms in literature such as Bloom Filters achieve similar goals.

In Step Q, we evolve the RCP queue length $q'_R(t)$ separately, instead of using the measured RCP queue, $q_R(t)$. The reason for this is TCP traffic is usually bursty and RCP's packets that get "stuck" behind TCP's appear to it as building up a queue. It does not necessarily mean that RCP is sending too many packets; they are simply blocked by TCP, which would not

happen if RCP traffic is isolated. Therefore, to achieve a fair rate, it is more reasonable to use $q'_R(t)$ to update RCP rate.

This algorithm introduces another parameter, the control update interval, T_c, whose value impacts the system stability. For example, if T_c is too small, RCP will reclaim the link-share it relinquished to TCP sooner than TCP has a chance to use it up, causing oscillations in $C_R(t), C_T(t)$ values. On the other hand, a large T_c value makes the algorithm sluggish, where RCP does not achieve its fair share. For stability, T_c should at least equal the amount of time needed for TCP to fill up the newly allocated link capacity to it; that is,

$$T_{c,min} = \frac{C_T - y_T(t)}{\dot{y}_T(t)},$$

where $y_T(t)$ is the input traffic rate of TCP, and $\dot{y}_T(t)$ the derivative of the traffic-rate.

An example of the deployment solution in action is demonstrated in Figure 5, in which there are equal RCP and TCP flows, and both receive their fair shares. The spikes correspond to TCP's saw-tooth behavior. When TCP backs off on a packet loss, RCP sees a room in link-rate and makes use of the spare capacity until TCP reclaims its link share. These spikes can be reduced by allocating a separate queue for RCP and isolating it from non-RCP traffic. These queues should be served with scheduling disciplines such as Deficit Round Robin or Weighted Fair Queueing.

More simulations in [10] show that this algorithm:

- Works under multiple bottleneck links to achieve equal bandwidth-sharing.
- Works for a mix of flow-sizes and RCP retains its property of short flow-completion times and its resemblance in performance to the ideal Processor Sharing scheme.

B. Hindrance 2: Coexisting with non-RCP bottlenecks

We cannot obviously have all routers and other network devices to be equipped overnight with RCP functionality. RCP end-hosts need to coordinate with non-RCP enabled network devices. If after the initial handshake, an RCP flow does not receive a valid rate, by default it switches to TCP congestion control since none of the routers along the path understand RCP. This situation is the simplest to resolve.

Even when an RCP flow receives a valid rate, it needs to find out if the router suggesting this rate is in fact the bottleneck. In this case, a flow optimistically starts by transmitting at the received rate feedback and uses a heuristic to figure out if in fact this is the bottleneck rate. Our proposed heuristic is based on an invariant property of RCP algorithm, in that when a queue builds up, RCP will react by reducing its rate to drain the queue. An end-host switches to TCP if it observes the following:

- Estimated round-trip time at end-host is doubled, or the number of outstanding segments are accumulated.
- There is no corresponding decrease in the received RCP rate from the network.

Once RCP determines that the received rate is not from the bottlenecked link, it has to switch to TCP congestion control immediately to avoid overwhelming the bottleneck.

The rationale behind our heuristic is: if the non-RCP bottleneck has a buffer size that equals bandwidth-RTT product (as most routers do today), then as the buffer fills up, the RTT will increase to at least twice the base RTT value and, if there is no corresponding reduction in RCP rate it is an indication to switch over. On the other hand if the buffering is smaller than bandwidth-RTT product, then packet losses occur sooner than the RTT doubles, indicating the need to switch. [10] has examples of this heuristic in action.

The heuristic works in the sense that there are no false-negative cases. When the network bottleneck shifts to a non-RCP router, an RCP endhost can always detect an increase in RTT and in the number of outstanding segments due to an invalid RCP sending rate suggested by non-bottlenecked RCP routers. However, there is still room for improvement in reducing the false-positive cases.

IV. Acknowledgments

We would like to thank Masayoshi Kobayashi, NEC Labs, Japan, for his ideas on Section III. We also thank Prof. Nick McKeown, Stanford University, for his invaluable feedback on this work.

References

[1] H. Balakrishnan, N. Dukkipati, N. McKeown, C. Tomlin, "Stability Analysis of Explicit Congestion Control Protocols," in *Stanford University Department of Aeronautics and Astronautics Report: SUDAAR 776*, Stanford University, September 2005.

[2] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, N. McKeown, "Processor Sharing Flows in the Internet," in *Thirteenth International Workshop on Quality of Service (IWQoS)*, Passau, Germany, June 2005.

[3] N. Dukkipati, N. McKeown, "Why Flow-Completion Time is the Right Metric for Congestion Control," in *ACM SIGCOMM Computer Communication Review*, Volume 36, Issue 1, January 2006.

[4] N. Dukkipati, N. McKeown, "Processor Sharing Flows in the Internet," in *High Performance Networking Group Technical Report TR04-HPNG-061604*, Stanford University, June 2004.

[5] N. Dukkipati, G. Gibb, N. McKeown, J. Zhu, "Building an RCP (Rate Control Protocol) Test Network," in *Hot Interconnects 15*, Stanford, August 2007.

[6] N. Dukkipati, N. McKeown, F. Baker, "Implementing RCP in the IPv6 Hop-by-Hop Options Header," *http://yuba.stanford.edu/rcp/*, Internet Draft (Work in Progress).

[7] D. Katabi, M. Handley, C. Rohrs, "Internet Congestion Control for High Bandwidth-Delay Product Networks," in *Proceedings of ACM Sigcomm*, Pittsburgh, August, 2002.

[8] I. McDonald, R. Nelson, "Congestion Control Advancements in Linux," in *linux.conf.au*, January 2006.

[9] T.J. Ott and T. V. Lakshman and L.H. Wong, "SRED: Stabilized RED" in *Proceedings of INFOCOM*, 1999.

[10] J. Zhu, C.H. Tai, N. Dukkipati, N. McKeown, "Making large scale deployment of RCP practical for real networks," in *High Performance Networking Group Technical Report TR07-HPNG-061807*, Stanford University, June 2007.

[11] "Rate Control Protocol (RCP) Home Page", *http://yuba.stanford.edu/rcp/*.

[12] "NetFPGA Home Page",*http://yuba.stanford.edu/NetFPGA/*.

[13] "The Network Simulator", *http://www.isi.edu/nsnam/ns/*.