Building a RCP (Rate Control Protocol) Test Network[*]

Nandita Dukkipati, Glen Gibb, Nick McKeown, Jiang Zhu[†]
Computer Systems Laboratory
Department of Electrical Engineering
Stanford University
{nanditad, grg, nickm, jiangzhu}@stanford.edu

Abstract

We recently proposed the Rate Control Protocol (RCP) as way to minimize download times (or flow-completion times). Simulations suggest that if RCP were widely deployed, downloads would frequently finish ten times 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 some scientific and repeatable experiments with RCP, we have built and tested an open and public implementation of RCP; we have made available both the end-host software, and the router hardware.

In this paper we describe our end-host implementation of RCP in Linux, and our router implementation in Verilog (on the NetFPGA platform). We hope that others will be able to use these implementations to experiment with RCP and further our understanding of congestion control.

1 Introduction and Motivation

Our goal is to enable others to perform repeatable scientific experiments with RCP (Rate Control Protocol), and to compare it with other proposed congestion control schemes. As with any congestion control mechanism, there are many aspects to be considered when deciding how good it is: Is it provably stable? Does it work well in simulation under a broad range of operating conditions? How well does it behave alongside TCP? How complex is it to implement? And how well does it work in practice?

In other work, the stability of RCP has been proved, and thousands of simulations suggest it is very promising under very broad conditions. In this work we are particularly interested in the last two questions: How difficult is it to implement RCP, and how well it works in practice with real users and traffic.

Before explaining how to implement it, 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). 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:

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

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, and α, β are parameters chosen for stability and performance.

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

[*]This research is supported by NSF under ITR award ANI-0331653, the Stanford Clean Slate Program, and NEC Corporation.

[†]Authors' names are in alphabetical order.

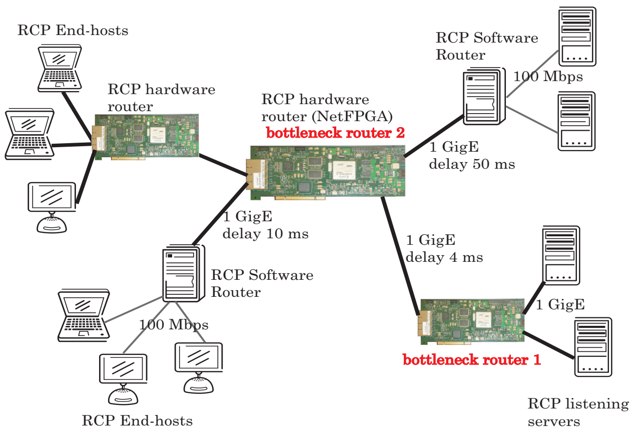


Figure 1. An example RCP test network with a mix of hardware and software routers, and RCP end-hosts, running over 100 Mbps/1 GigE links.

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 and XCP, 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 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.

RCP is described in detail in [1], which describes the motivation behind RCP and why flow-completion time is the appropriate metric for congestion control. The RCP protocol, mechanisms, algorithm and simulations showing short flow-completion times are described in [2] [3]. And [4] uses control theory to show that RCP is stable independent of the link-capacities, number of flows and network round-trip times.

Although we have many thousands of promising $ns2$ simulations, we want to find out how well RCP performs in practice; and how complex it is. We are particularly interested in how complex the changes are to the routers. Router vendors are, understandably, very reluctant to add new features to the forwarding path of their routers, particularly if they involve complex calculations. Routers are already overloaded with many features, and are limited by the power they consume. Great care needs to be given to bloating the requirements further. So in our work, we try to analyze the additional complexity of the router.

Once we have a full implementation of RCP, we can build a test network and see whether it behaves as promised. To make it realistic, we want to use real implementations of RCP end-host and routers, real hardware and links, and realistic topologies. Figure 1 illustrates an example of an RCP test network with a mix of hardware and software routers, as well as RCP end-hosts on Linux, all connected through 100 Mbps or 1 GigE links. This paper describes our efforts in building one such testbed, in particular we describe the design and implementation of each of the testbed components - RCP hardware/software router and the end-hosts - along with a description of some preliminary experiments.

The rest of this paper is arranged as follows: Section 2 describes our goals and nature of experiments we plan to run on the test network, Section 3 describes RCP end-host implementation, RCP router implementation is described in sections 4 (hardware) and 5 (software), we quantify the implementation complexity in Section 6, and go on to describe some preliminary experiments in Section 7.

2 Nature of Experiments we want to run on RCP Test Network

We want two kinds of experiments:

1) Experiments to test the RCP router and end-host implementations: These experiments will use deterministic traffic, for example a set of long-lived flows that have specific start and end times. The goal is to observe the RCP system behavior such as how the RCP rate varies at the routers, the evolution of queue-sizes at bottlenecked links, and other statistics.

2) Experiments that demonstrate RCP's short flow-completion times and stability under traffic representative of that in the Internet: These experiments will study RCP's performance under more complex traffic patterns and network topologies[1]. The goal is to verify and explore RCP properties - its strengths such as short flow-completion times and stability for a wide range of network and traffic characteristics, as well as its weaknesses such as transient queue spikes and buffer overflows.

In the next two sections, we describe the two main components of RCP test network, the end-host and router.

[1]Including dynamic traffic patterns (such as different flow arrival patterns, flow-size distributions, offered loads, multiple congested bottleneck links, reverse path congestion, etc.), that represent a work load mix of web browsing and file downloads.

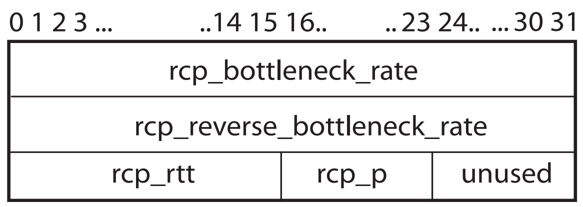


Figure 2. The 12-Byte RCP header: the *rcp_bottleneck_rate* **(4 Bytes) carries the rate (in Bytes/msec) of the most congested link along the path;** *rcp_reverse_bottleneck_rate* **(4 Bytes) is the bottleneck rate (in Bytes/msec) echoed by the receiver, so the sender can adapt its rate;** *rcp_rtt* **(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.**

3 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.

3.1 Placement and Format of RCP Header

RCP is implemented as its own protocol layer between IP and transport layers, as shown in Figure 3. Other places to carry RCP information would be IP or TCP options, each having its pros and cons[2]. 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, with the following four fields:

[2]Recent studies [6] have shown that 70% of connections are not established when SYN segments have a new IP option X and, a third of connections not established even for known IP options. Further, packets with IP options take the slow-path on routers. TCP options on the other hand are more widely used and the same study showed only 0.2% connections failed on introduction of new TCP option. The downside is that routers would need to modify TCP header and be aware of every new transport protocol that uses RCP information.

1. *rcp_bottleneck_rate* carries the bottleneck rate of the most congested link along the path. The RCP sender puts in a zero here - meaning the sender would like as high a rate as the network gives it. Alternatively, it can also put in the local interface rate. The routers overwrite this rate as it passes through the network.

2. *rcp_reverse_path_bottleneck_rate* is filled by RCP receiver to communicate the bottleneck rate to an RCP sender.

3. *rcp_rtt* carries the sender's round-trip time estimate and is used by routers to update their traffic-averaged RTT estimate.

4. *rcp_p* is higher layer protocol number such as that of TCP, UDP.

All Rates are expressed in Bytes/msec, and RTT in msecs.

3.2 RCP End-host Functions

This section describes the RCP implementation at sender and receiver end-hosts. We will focus here on the case of TCP transport protocol running over RCP. Figure 3 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 [7], 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.

The 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

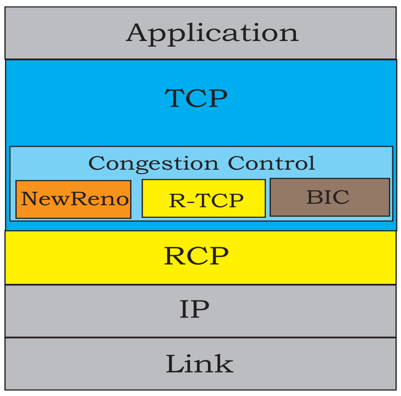
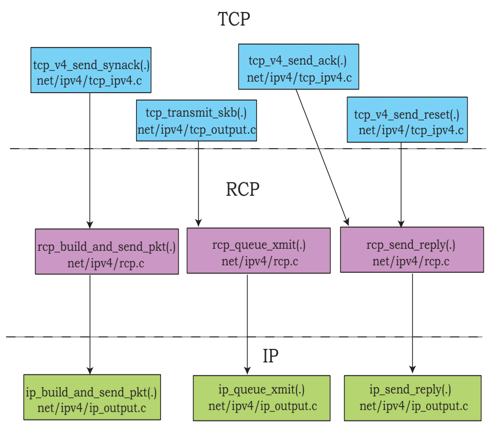


Figure 3. RCP is a protocol between the IP and transport layers.



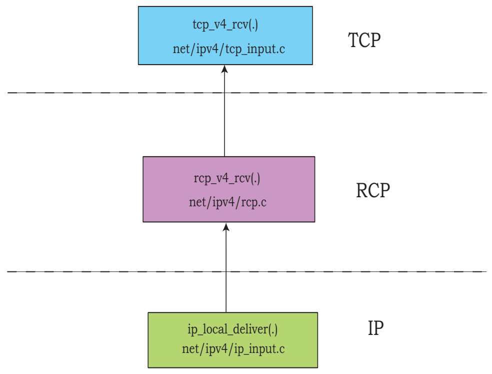
TCP
tcp_v4_send_synack(.) net/ipv4/tcp_ipv4.c
tcp_v4_send_ack(.) net/ipv4/tcp_ipv4.c
tcp_transmit_skb(.) net/ipv4/tcp_output.c
tcp_v4_send_reset(.) net/ipv4/tcp_ipv4.c
RCP
rcp_build_and_send_pkt(.) net/ipv4/rcp.c
rcp_queue_xmit(.) net/ipv4/rcp.c
rcp_send_reply(.) net/ipv4/rcp.c
IP
ip_build_and_send_pkt(.) net/ipv4/ip_output.c
ip_queue_xmit(.) net/ipv4/ip_output.c
ip_send_reply(.) net/ipv4/ip_output.c

Figure 4. Data path of outgoing RCP packets. RCP intercepts the function calls from TCP to IP, to introduce its 12-Byte RCP header and fill in the rate and RTT fields.

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. Figure 4 shows the paths that different packets take from TCP to the lower RCP layer. SYN, RESET, DATA, and ACK packets take different paths, and RCP intercepts all calls from TCP to IP, to attach the 12-Byte RCP header. The incoming RCP packets from the network have only one path up the stack, where RCP intercepts the function calls from IP to TCP, to strip of the 12-Byte header and pass the segment to TCP, as shown in Figure 5.

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



tcp_v4_rcv(.) net/ipv4/tcp_input.c
TCP
rcp_v4_rcv(.) net/ipv4/rcp.c
RCP
ip_local_deliver(.) net/ipv4/ip_input.c
IP

Figure 5. Data path of incoming RCP packets. RCP intercepts the function calls from IP to TCP, to strip of the 12-byte header and pass the segment to TCP.

congestion window as shown below, overriding the slow-start and congestion avoidance window changes:

```
snd_wnd = (rcp_bottleneck_rate * rcp_rtt) /
  (MSS + RCP_HEADER_SIZE + IP_HEADER_SIZE)
```

where MSS is the maximum segment size. Just as other flavors of TCP, *R-TCP* uses a window which is the minimum of above window calculation and receiver advertised window. It also maintains the window size to be least equal to one Maximum Segment Size. *R-TCP* keeps track of the smoothed round-trip time estimate for this connection.

The sender paces packets from TCP's send queue at the following pacing interval:

```
packet_pacing_interval = MSS/rcp_bottleneck_rate;
```

The mechanisms that decide which packets to retransmit upon losses remain the same as in TCP.

4 RCP Router based on NetFPGA

This section outlines the implementation of an RCP router in hardware, based on the RCP description and specification in [2] and [13]. Such an implementation demonstrates the feasibility and simplicity of supporting RCP within a router. We begin by describing the operation of a non-RCP router design before elaborating on our RCP design.

4.1 Vanilla router functionality

The operation of a router can be subdivided into two parts – the data path and the control path.

The data path processes incoming packets and routes them towards their destination. Tasks performed on each

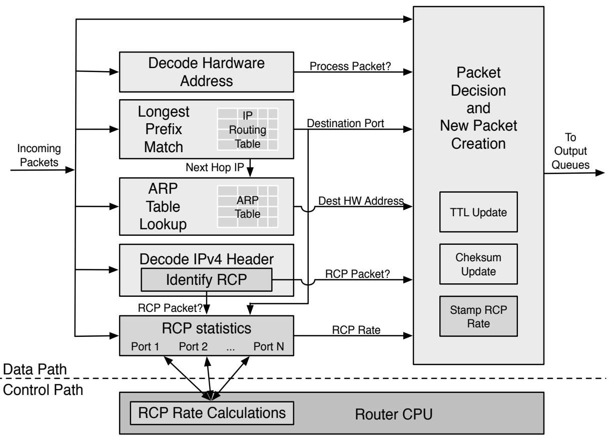


Figure 6. A generic hardware router with RCP support

packet include verifying the IP header checksum, extracting the destination address from the IP header, performing a longest prefix match look-up of the address in the routing table, decrementing the packet's time-to-live, updating the checksum, and forwarding the updated packet out the correct interface. The data path is implemented in hardware for speed since it needs to process every packet.

The control path is responsible for a set of tasks that are performed infrequently such as maintaining the routing tables and providing a control interface to the router. Since operations on the control path occur relatively infrequently they are usually implemented in software and executed on a CPU inside the router.

4.2 RCP router enhancements

An RCP-enabled router must additionally compute the fair-share rate (as per Equation 1) and stamp that rate into the headers of RCP packets. The rate computation requires the router to maintain an average round-trip time estimate for outgoing traffic on each interface using the RTT information carried in RCP packets. The rate is computed once every control interval which is in the order of a round-trip time. Statistics (aggregate incoming traffic and average RTT) are gathered during each control interval which are then used for the rate computation. When an RCP packet arrives, the router adds RTT value of packet header to the running sum it maintains and before departure the packet is stamped with the RCP rate.

As with the vanilla router functionality, the RCP functionality is split between the hardware data path and the software control path. Figure 6 shows the RCP enhancements to a vanilla router implementation. The additions to the data and control paths are summarized below:

1. Per-packet data-path processing in hardware:

   - Identifying whether an incoming packet is an RCP packet (RCP Identification)
   - Updating a running RTT sum of the outgoing interface, if the packet carries a valid RTT (RCP stats per-port)
   - Updating the aggregate traffic destined to the outgoing interface (RCP stats per-port)
   - Stamping the RCP rate in the outgoing packet (Stamp RCP Rate)

2. Periodic control path computations in software:

   The following are calculated approximately once per average RTT of traffic transiting the router:

   - The bandwidth, $R(t)$, allocated to the average data flow as per Equation 1.
   - The moving round-trip time average, as detailed in [13].

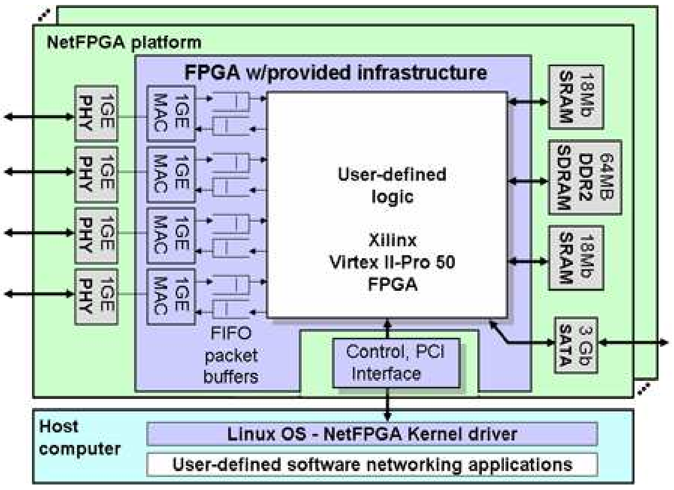


Figure 7. A block diagram of the NetFPGA hardware platform. The platform consists of a PCI card which hosts a user-programmable FPGA, SRAM, DRAM, and four 1 Gbps Ethernet ports.

Our hardware implementation utilizes the NetFPGA programmable hardware platform. NetFPGA is a programmable hardware platform for network teaching and research [10][11][12]. The platform consists of a PCI card, which hosts a user-programmable FPGA, SRAM, DRAM, and four 1 Gbps Ethernet ports, together with associated software for programming and control. A block diagram of the platform is shown in Figure 7.

The remainder of this section describes the per-packet processing and the periodic computations in more detail.

4.3 Per-packet processing in hardware

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 in the data path hardware. Pseudo-code of the data path operations, listed below, clearly illustrates the modest processing requirements. Upon packet arrival the router must update counts for the corresponding output port of the running RTT sum, the number of arriving bytes, and the number of packets carrying a valid RTT. On packet departure the router overwrites the bottleneck rate carried in the packet if need be.

Processing performed upon packet arrival

```
input_traffic_Bytes += packet_size_Bytes
if (this_packet_RTT < MAX_ALLOWABLE_RTT)
  sum_rtt_Tr += this_packet_RTT
  num_pkts_with_rtt += 1
```

Processing performed upon packet departure

```
if (packet_BW_Request > rcp_rate)
  packet_BW_Request = rcp_rate
```

A quantification of the implementation complexity as well as achievable clock rates can be found in Section 6.

4.4 The control path in software

The control path performs periodic rate and RTT computations, and since these are performed infrequently they are implemented in software. In our implementation this software runs on the host in which the NetFPGA board is installed – in practice this would run on the CPU of the router.

The control path only performs work at initialization when each port of the RCP router is brought online and at the expiration of each timeslot. The initialization performed by the control path simply sets the various RCP parameters within the router to their default values. Upon expiration of a timer the control path reads the hardware registers to retrieve the RCP statistics, performs the necessary rate and RTT computations, writes the updated RCP rate and timeslot interval to the hardware registers, and then restarts the timeslot timer.

It should be noted that *all* multiplication and division operations required for rate calculations are performed within the control path. This allows RCP to take advantage of the multiplication/division operations of the router's CPU.

The control path software interfaces with the hardware via the registers provided by the hardware. The number of registers required per port is small – we provided 7 registers per-port for input-traffic, queue occupancy, current timeslot duration, elapsed time within the current time-slot, RCP rate to be stamped into the packet, sum of RTTs in packets, and input traffic in bytes carrying valid RTT.

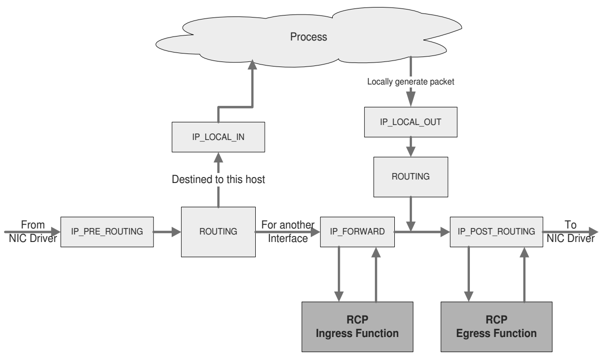


Figure 8. Linux NetFilter: Packet processing paths and the points where RCP is hooked.

5 RCP Router based on commodity Linux System

We have also implemented a software version of RCP router that can run on any commodity hardware running Linux. Our RCP software router is 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.

Similar to RCP router on NetFPGA, RCP soft router consists of a control plane and data plane. Control plane is a timer driven function to update 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 the packet processing chain in kernel. The operations consist of Ingress and Egress functions and are similar to those described in Section 4.3 :

• 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 (only when packet carries valid RTT); it also updates the aggregate traffic rate to the outgoing interface.

• Egress function is registered with IP_POSTROUTING hook in NetFilter. When a packet is ready to go to one of the outgoing interfaces after routing, this function stamps RCP rate in the header and updates the TX queue occupancy for that interface.

Figure 8 shows the packet processing chain in NetFilter and where exactly the RCP functions are hooked up.

6 Quantifying the Implementation Complexity

6.1 Complexity of RCP NetFPGA implementation

RCP takes 31,000 gates of the 4.1×10^6 gates used for the IP router implementation on the NetFPGA platform, amounting to 0.75% of the total logic used. This translates to a die area of $0.1 - 0.2\ mm^2$ in a $90nm$ ASIC. Clearly, RCP is simple to implement in high-speed routers.

Our reference router (running on a Xilinx Virtex II Pro 30) used a core clock frequency of 62.5 MHz. The RCP implementation was not the bottleneck in this design.
Hardware resources:

- $5 \times$ 32-bit software accessible registers per port
- $2 \times$ 16-bit software accessible registers per port
- $2 \times$ 16-bit counters per port (used by timers)
- $2 \times$ 32-bit adders per port (packet/queue statistics)
- $1 \times$ 32-bit counter per port (packet/queue statistics)

Lines of code (including commenting, data structure declarations, etc.):

- Verilog (data path): approximately 600 lines
- C (control path): approximately 350 lines

6.2 RCP Software Router and End-host

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 1.

Table 1. Complexity of RCP software router

	Control	Data Plane		
	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 software router that is running linux kernel without RCP support and that with our RCP data plane, we measured the time it takes a packet to traverse through the IP forwarding path for both cases. For a non-RCP enabled Linux kernel, each packet processing takes 9.7368 jiffies, while with the RCP data plane enabled it takes 9.9998 jiffies. Therefore, RCP-related processing on the software router is only 2.6% of the total processing for IP packet forwarding in Linux kernel.

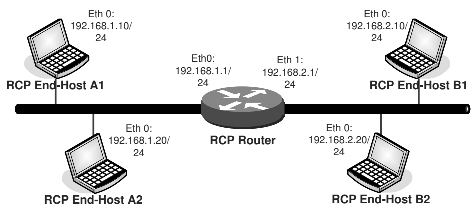


Figure 9. A simple experiment topology

7 Experiments

As an initial step to test and validate our implementation correctness, we used User Mode Linux, UML [14], for both RCP end-host and router in a virtual machine environment. We used a single Linux machine running multiple UML instances, connected via virtual Giga-Ethernet to form certain topologies. The second approach is to use real equipment running our RCP end-host software and RCP NetFPGA router with real Gigabit switches connecting them into certain topologies.

We start four UML instances running RCP end-host software and one UML instance running RCP router as shown in Figure 9. On one side of the network, we use iperf [15] to generate multiple continuous traffic flows while on the other end, we use iperf server to receive flows and collect statistics. The RCP router in between has IP routing turned on; RCP-router-driver watches for RCP traffic, and if necessary rewrites the RCP rate in the packets flowing through.

In a typical experimental scenario, the end-host stations start flows and stamp an initial starting rate $R1$ in RCP packets; the RCP router converges to the correct fair-share rate (which in case of a single flow would be the capacity of links on the RCP router). At the router, we monitor the RCP rate and traffic-averaged RTT value, while at the end-hosts we keep track of the congestion window-size, and sender's RTT estimate. Figure 10 illustrates the results from one such experiment.

8 Conclusion

We achieved the goals we had started out with – we now have a working implementation of RCP router and end-host but most importantly, we demonstrated how it is extremely simple to implement RCP in router hardware. Coupled

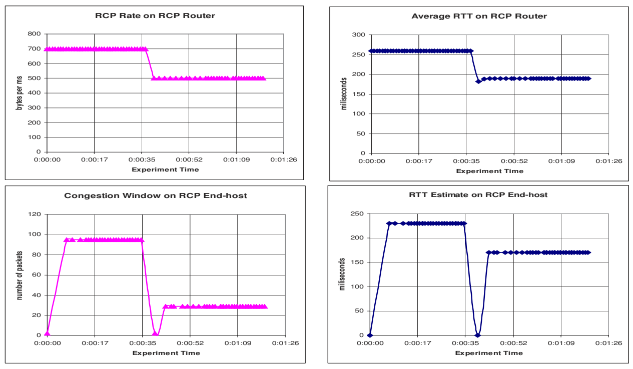


Figure 10. RCP Experiment Result: The RCP router was configured to allow a maximum rate of 700 packets/milisecond initially. When the traffic started to flow, we adjusted the rate to be 500 packets/milisecond to monitor how the RCP router and end-host would react to this change. The above plots show that the RCP router rate dropped from 700 to 500 pkts/msec (top left plot) and the end-host adjusted the congestion window from 95 to 30 pkts (bottom left plot) after the change.

with our previous results on RCP's performance, the present work gives us the confidence that RCP congestion control achieves short flow-completion (close to ideal processor-sharing) *while doing very little work at the routers*.

We are now ready to use our implementation and build a RCP test network. We will use these implementations to find out how well RCP performs in practice with real hardware, real links, and realistic topologies and traffic.

References

[1] 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.

[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, ”Processor Sharing Flows in the Internet,” in *High Performance Networking Group Technical Report TR04-HPNG-061604*, Stanford University, June 2004.

[4] 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.

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

[6] A. Medina, S. Floyd, M. Allman, ”Measuring Evolution of Transport Protocols in the Internet,” in *ACM Computer Communications Review*, April 2005.

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

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

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

[10] M. Casado, G. Watson, N. McKeown, ”Reconfigurable Networking Hardware: A Classroom Tool,” in *Hot Interconnects 13*, Stanford, August 2005.

[11] M. Casado, G. Watson, N. McKeown, ”Teaching Networking Hardware,” in *ITiCSE, Monte de Caparica*, Portugal, June 2005.

[12] ”NetFPGA Web Page”,*http://NetFPGA.org*.

[13] 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).

[14] ”User Mode Linux”, *http://user-mode-linux.sourceforge.net/*.

[15] ”Network Performance Measuring Tool: iperf”, *http://dast.nlanr.net/Projects/Iperf/*.