RackCC: Rack-level Congestion Control

Danyang Zhuo, Qiao Zhang, Vincent Liu[†], Arvind Krishnamurthy, Thomas Anderson
University of Washington, University of Pennsylvania[†]
{danyangz,qiao,vincent,arvind,tom}@cs.washington.edu

ABSTRACT

Many data center traffic patterns exhibit abundant concurrent connections and high churn. In the face of these characteristics, server-centric congestion control is a poor fit—each connection, no matter how small, must start from scratch testing when and how much to send along a given path. This is despite the fact that there are a large number of flows that may have already probed the exact same path, not just at a server level, but also at a rack level. Thus, we argue for *rack-level congestion control* in which all connections are tunneled through rack-to-rack JumboFlows. This design allows an entire rack's connections to cooperate with one another for better fairness and performance, particularly for short flows.

In this paper, we examine situations in which JumboFlows might be useful and present a preliminary design of a system (RackCC) that implements JumboFlows.

1. INTRODUCTION

At any given time, a server in a large data center might have up to 100s or 1000s of concurrent connections [4, 16] most of which are short-lived and bursty. In the face of abundant flows and churn, TCP is a poor fit: every connection operates in a bubble and ignores the congestion information gleaned by previous and current flows. While TCP is simple, well-understood, and able to eventually converge to max-min fairness, the pathological nature of some data center traffic patterns expose several inefficiencies:

- *Long ramp-up times:* Every new connection begins with a clean slate. In TCP, this involves starting at a (somewhat arbitrary) fixed initial window and adjusting up/down to the desired throughput over the course of several RTTs. Short-lived connections suffer from these ramp-up times.
- *Uneven bandwidth allocation:* Connections that experience loss or explicit congestion notification (ECN) are each independently responsible for decreasing their own load in the network. In a steady state, all flows are throttled equally through randomness. However, if the flows are short, affected connections are inordinately penalized.
- *Disjoint behavior:* Because connections operate independently of one another, aggregate behavior can be problematic. The classic example of this is TCP incast, where many connections, oblivious of one another, send a synchronized burst of packets to single destination that overloads the network and causes high loss rates and timeouts.

All of these issues get worse as network bandwidth continues to grow while per-core CPU performance plateaus. This is in addition to the power-efficiency-based push for more and more CPUs per rack [15]. In either case, the end result is more connections per path.

In this paper, we make a simple observation: that for any new connection on any given network path, a great deal of connections have already probed the *exact* same path, not just at a server level, but at a rack level as well. Therefore, we argue that many data center deployments would benefit from connection pooling at the level of entire racks.

Recent measurement studies suggest that such a level of aggregation may be effective for many deployments. For instance, many data centers have a fair degree of locality within clusters [16, 17] and sometimes racks [6]. In these cases, as much as 75% of a rack's flows end at the same destination server [16]—a number that may be higher if we aggregate destinations at a rack level. This is despite some existing server-level connection pooling. Even in multi-tenant data centers, the set of communication partners of any given server or rack is only a small subset of the entire data center [9].

We propose to tunnel connections that share a source and destination rack through a single, long-running, ToR-ToR, TCP-like *JumboFlow*. In a sense, the JumboFlow allows us to make use of TCP's congestion control protocol for that which it was designed: ensuring fair share and good performance among long-running flows.

In addition to improving aggregate rack-to-rack throughput, rack-level congestion control also addresses the issues introduced above. Much like a SPDY [1] connection, a Jum-

HotNets-XV, November 09-10, 2016, Atlanta, GA, USA
© 2016 ACM. ISBN 978-1-4503-4661-0/16/11…$15.00
DOI: http://dx.doi.org/10.1145/3005745.3005772

boFlow improves the ramp-up time of short flows by virtue of the JumboFlow being more persistent. Within a JumboFlow, new subflows can immediately take their share of the current congestion window, eliminating long startup times and facilitating fairness. In addition, when a loss occurs, all subflows back off an equal amount.

The primary challenge in designing such a system is one of coordination: servers must be able to cooperatively arrive at a congestion window that responds to each and every new ACK and subflow. This information sharing must have low communication overhead and should operate without requiring complex logic or massive buffers at the ToRs.

Our primary contribution is to show that such a system is feasible with limited switch support in the form of a small amount of additional counters and state. To this end, we present a preliminary design of a system, RackCC, that facilitates coordination between flows within a rack. RackCC is able to emulate the aggregate behavior of an entire rack's TCP connections, but with more fairness within the rack. Thus, our design is TCP-friendly with legacy racks.

2. THE CASE FOR RACKCC

A typical data center network is highly structured. Tens to hundreds of thousands of servers are stacked in physical racks. In each rack or small set of racks, servers connect to a single Top-of-Rack (ToR) switch, which serves as their gateway to the rest of the network. The ToRs serve as leaves in a multi-rooted, multi-level tree interconnection network.

Traditional congestion control protocols ignore the structured layout of servers in the network. From the many variants of TCP [11] to more recent protocols like CUBIC [10], DCTCP [4] to network-assisted protocols like RCP [8], and XCP [13], each of these proposals treats every flow as independent. In a network with unpredictable traffic patterns, this promotes simplicity and effective handling of any usage scenario. In a highly-engineered, high-performance data center, it can also lead to inefficiency.

To see why, imagine we have an existing TCP connection between two racks. Assuming the connection is more than a few RTTs old, it will be much closer to its optimal rate than a flow that is just starting out. When a server creates a new connection between the same two racks, traditional TCP would ignore the congestion information gathered by the existing flow. Instead, the new flow would re-probe the path's bandwidth from scratch. If flows are short or arrive frequently, throughput will suffer.[1]

When a drop does occur, only a single connection slows down while others on the same path continue to speed up. Ideally, all of the flows that share the bottleneck will slow down equally. While TCP can achieve this using randomness over long time periods, convergence is again often too slow for short flows, which contributes to higher tail latencies for the few connections that experience losses.

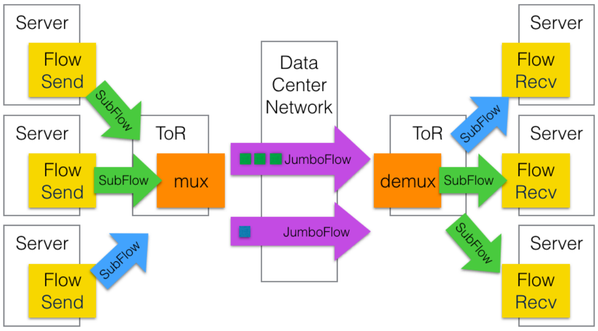


Figure 1: RackCC Architecture. ToRs keep several consistent JumboFlow connections to other ToRs. Applications start subflows that are multiplexed into one of the JumboFlows. Subflows inside the same JumboFlow share congestion information with each other.

Our basic approach in this paper is to aggregate congestion control among flows that traverse the same ToR-to-ToR path. Note that rack-level congestion control may not be effective for all data center deployments. Indeed, the efficacy of this approach is heavily dependent on the amount of rack-level path sharing and workload of the system. However, we do note that there are at least some deployments for which this approach will be useful. A prime example is a data center where operators collocate machines with similar purposes within the same racks. For instance, if several racks are set aside for data analytics (e.g., MapReduce), one would expect a great deal of redundant rack-level paths between racks.

3. RACKCC DESIGN

RackCC is a system that implements rack-level congestion control using ToR-ToR JumboFlows. Our design is motivated by three main goals:

1. *Fast flow start:* when a new flow begins, it should utilize existing information about the congestion of the network. In particular, existing flows should make room for the new flow to take its fair share immediately. A fast flow start avoids costly multi-RTT bandwidth probing.
2. *Fate-sharing of flows:* When congestion occurs, all subflows on the same rack-to-rack path should slow down simultaneously. This promotes fairness among flows.
3. *Collaborative congestion detection:* ToR-to-ToR congestion should be detected and handled collaboratively by all members of a JumboFlow. Detection is more accurate as a result, alleviating problems such as TCP incast.

In RackCC, distributed applications still create individual connections between one another, but then wrap these subflows within ToR-ToR JumboFlows. Each endpoint will coordinate with the other endpoints sourcing subflows within the same JumboFlow to agree on fair sending rates. Essentially, each JumboFlow contains one or more subflows, and multiple JumboFlows connect each communicating rack. A high-level diagram of this architecture is illustrated in Figure 1.

[1] Note also that maintaining persistent TCP connections between communicating endpoints doesn't help as the TCP slow start mechanism kicks in for data sent over idle connections.

Note, however, that we do not change the TCP interface—coordination between endpoints is entirely transparent.

Note that our proposal only seeks to aggregate congestion control. TCP has many other features that are not included in this. Servers are therefore still responsible for detecting packet drops and retransmitting dropped packets.

Given a JumboFlow-based architecture, there are two questions we must answer: (1) how do we set the rate of a JumboFlow in a way that is simple to implement and scales well to high flow counts? and (2) how do subflows share a JumboFlow given that their bottlenecks and demands may vary? Our design borrows ideas about congestion control from existing TCP variants like DCTCP [4] as well as bandwidth allocation from proposals like EyeQ [12] and RCP [8], but composes them in a new and interesting way.

In the remainder of this section, we describe our preliminary system design. We begin by discussing how to implement congestion control for JumboFlows in the absence of varying demand and packet loss at the server-ToR access links. We then describe how subflows split the bandwidth provided by their parent JumboFlow.

3.1 JumboFlow Congestion Control

As much as possible, we want the aggregate behavior of an N-subflow JumboFlow to emulate the aggregate behavior of N DCTCP [4] flows. The choice of DCTCP is deliberate—several of its design decisions are particularly amenable to rack-level coordination.

Of particular note is DCTCP's reliance on Explicit Congestion Notification (ECN)—a mechanism in which intermediate switches notify sources of congestion in the network by explicitly marking packets—as the primary indicator of congestion. RackCC relies on this same indicator for controlling JumboFlow congestion. Upon receipt of a packet, every switch except the last hop (i.e., the destination's ToR switch) will check the queue occupancy of the outgoing link. If the queue has more than K bytes of data, the new packet is marked. Destination servers will echo back this marker to inform the source rack of congestion. This mechanism is a good fit for rack-level congestion control because keeping track of ECN rates requires very little state in the switch as opposed to keeping a record of packets to track loss rates.

Determining the aggregate rate of a JumboFlow. Before we discuss how we utilize ECN to set the rate of individual subflows, we first describe the desired aggregate behavior of a JumboFlow. Assume we have N subflows of which N' are bottlenecked by their share of the JumboFlow (as opposed to being bottlenecked by the source-ToR or ToR-destination access links). In each round, some number of packets are marked. The source ToR maintains an exponentially-weighted moving average, α, of the fraction of packets that are marked. The desired aggregate rate for the next RTT is then:

$$F_{target} \leftarrow \begin{cases} F_{cur} + N'r, & \text{if } \alpha = 0 \\ F_{cur} \times (1 - \frac{\alpha}{2}), & \text{if } \alpha > 0 \end{cases}$$

In the first case, when there are no marked packets, r is scaling factor that determines the aggressiveness of the congestion control algorithm. To make JumboFlow fair to standard DCTCP, we assume $r = \frac{\text{MSS}}{\text{RTT}}$.

Implementing JumboFlows. To ensure that servers obey the above target rate, most of the heavy lifting is done on the end hosts. In fact, ToRs in our system only need to implement a simple counter-based interface that is similar to functionality that already exists. In particular, we assume the ToR keeps track of the counters in Table 1 and broadcasts them to its servers at frequent intervals.

Using the value of α provided by the ToR, servers set a subflow's congestion window using the following equation. Note that 'unconstrained' means that the subflow is not bottlenecked by either access link. We describe how servers discover this property in Section 3.2.

$$f_{target} \leftarrow \begin{cases} f_{cur} + r, & \text{if } \alpha = 0 \text{ \& unconstrained} \\ f_{cur}, & \text{if } \alpha = 0 \text{ \& constrained} \\ f_{cur} \times (1 - \frac{\alpha}{2}), & \text{if } \alpha > 0 \end{cases}$$

This differs from DCTCP in a few ways: (1) α represents the ECN rate on the ToR-ToR path only, thus it does not take into account congestion on access links (2) α is a more accurate measure of congestion on the ToR-to-ToR path because it is taken across many subflows, (3) α is applied across all subflows equally. All of these contribute to faster, more fair convergence to max-min fairness on the ToR-ToR path.

3.2 Subflow Congestion Control

In the previous subsection, we described how individual subflows react to inter-ToR congestion information to emulate many DCTCP connections. However, these subflows are not all equal as different subflows may be bottlenecked by different access links or have different demands. In general, this problem requires multiple rounds of global coordination since subflows from different JumboFlows can compete for the same access links. Our solution is to have end hosts iteratively negotiate each subflow's fair share using mechanisms similar to that of RCP [8] or EyeQ [12].

To this end, each server has a sender module and a receiver module. The server's sender module is responsible for setting the sending rate of every subflow that originates from it. To do so, sender modules maintain a running estimate of the approximate rate at which local applications are offering traffic. They then compute the ideal rate by negotiating with connected receiver modules as well as the source's ToR.

Determining the ideal rate for each subflow. At any given time, a subflow's share of the ToR-ToR path is f_{target}. If we assume that JumboFlows effectively handle congestion in the ToR-ToR path, then the subflow can ignore congestion in the middle of the network, provided it stays under f_{target}. The subflow's ideal rate is thus primarily dependent on its fair share of the source and destination link. We handle these two bottlenecks differently.

Notation	Definition	Section
α	An exponentially-weighted moving average of a JumboFlow's ECN rate.	3.1
M	The number of new subflows.	3.3
N	The number of registered subflows.	3.3
T	The average total throughput.	3.3

Table 1: The per-JumboFlow state maintained by each JumboFlow's source ToR.

Destination. The destination periodically advertises a rate D_{target} to all connected sources. This rate is recursively updated periodically using an RCP/EyeQ-like formula:

$$D_{target} \leftarrow D_{cur}(1 + \frac{k \cdot (C-y) - k' \cdot \frac{Q}{d}}{C})$$

where d is the control interval, C is the link capacity of ToR-server access link, y the measured throughput in the last control interval, Q is the persistent queue size and k, k' are stability constants. The destination server knows all these variables except Q; the ToR needs to periodically communicate Q to its servers. The intuition behind this formula is as follows. If the destination is underutilized, the residual capacity $(C-y)$ will be high and the persistent queuing term $\frac{Q}{d}$ will be low, resulting in an increase in D. If the destination is congested, $(C-y)$ will approach zero while persistent queuing rises, resulting in a decrease in the target rate.

Note that the destination broadcasts the same value of D_{target} to all connected sources regardless of their demands and bottlenecks. The control loop efficiently arrives at max-min fairness by bounding the maximum rate of any subflow—all other flows proceed at their fastest rate.

Source. For a given subflow, the source now has both f_{target}, the subflow's share of its JumboFlow, and D_{target}, the destination's maximum allowable rate. From these two values, the source will set the subflow's maximum allowable rate:

$$s_{target} \leftarrow \min(f_{target}, D_{target})$$

Now armed with s_{target} for every one of its subflows, the source will decide which packets to send next using fair queuing and a rate limit of s_{target} for each subflow. This mechanism ensures max-min fairness, regardless of each subflow's demands and bottlenecks, and it is performed at the server without requiring switch support. As a concrete example, consider a case where a source with a 10 GbE connection has three subflows that have s_{target} values of 2, 6, and 10 Gbps. Fair queuing will rotate between the three such that each subflow receives 2, 4, and 4 Gbps, respectively.

Recall that in Section 3.1, we assumed that sources could determine whether they were bottlenecked at an access link or by the JumboFlow. With the above protocol, this information is easy to derive. If the final, observed rate of the subflow is f_{target}, it is bottlenecked by its JumboFlow. Otherwise, it is not and the source should not increase f_{target}.

3.3 Fast Start and Fast End

Now that we have described how connections adjust their sending rate to account for congestion in either a JumboFlow or an access link, we now discuss what happens when connections first start up or end.

In traditional TCP, new connections start at some small, initial window size and slowly ramp up from there. In RackCC, we instead attempt to jump start new flows to their fair share right away. In essence, we prioritize new flows over existing ones because new flows are more likely to be short. To implement this, both the access links and the JumboFlow need to make room for the new subflow.

Access Links. When a new flow begins, the source and destination servers both function normally. The destination, upon receiving a SYN packet, will simply return its existing D_{target} value. The source, upon receiving D_{target} and its initial f_{target} (described below), will utilize fair queuing exactly as before. A flow end is handled similarly: the destination relies on its control loop to manage D_{target}, while the source removes the subflow from its fair queue.

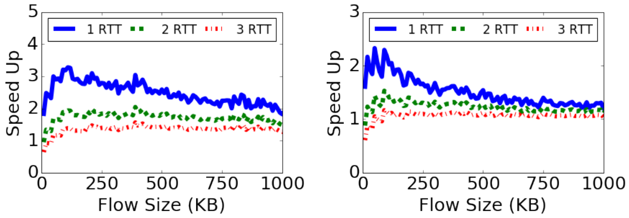
For both ends of the connection, high churn is fine as long as the churn involves similarly-sized flows. If there is a sudden burst or dearth in flow quantity and size, there will be temporary congestion, but as in RCP, the congestion control protocol will quickly converge to address the issue.

JumboFlows. For the JumboFlow, the source's ToR can detect the beginning of a new connection by capturing the destination's initial SYN response and/or D_{target} advertisement. Its goal at that point is to give the new subflow its fair share of the JumboFlow and reduce the share of the other subflows. More specifically, when M new subflows join the JumboFlow:

1. Each new subflow should get $\frac{1}{N+M}$ of the JumboFlow. This means that each of the N existing flows should back off by a factor of $\frac{M}{N+M}$.
2. Inform the source of its initial f_{target}: $\frac{T}{N+M}$.
3. $N \leftarrow N + M$.

When a connection ends, the server must unregister the subflow in the ToR to decrease N by 1. Even with high churn and very short flow sizes, JumboFlows remain relatively stable. Servers do not need to be informed of every update immediately and can handle them in batches. In addition, adding/removing flows is efficient because servers only need to query and account for M, N, T alongside its queries for α rather than needing to have a separate process for every connection start and stop.

Note that a side effect of this scheme is that it encourages short flows. Long-running, bursty connections are in-



(a) Num of Subflows = 2

(b) Num of Subflows = 10

Figure 2: Comparison of RackCC's Fast Start and TCP's slow start. This graph plots the average speedup of a new flow placed in the midst of many existing flows. Results are shown for a range of fixed new flow sizes, number of existing subflows, and Fast Start communication latency.

centivized to register and unregister themselves depending on demand. Doing so gives them the same performance benefits as short connections in RackCC. This is in contrast to traditional TCP, where idle connections must fall back into slow start every time. Also note that this process does not necessarily involve the OS—registering and unregistering is a purely transport-layer concern.

4. EVALUATION

To evaluate the efficacy of RackCC in addressing the inefficiencies of traditional TCP protocols, we a combination of analysis and packet-level simulations. There are two questions we sought to answer:

- Can RackCC quickly ramp-up flows?
- Does sharing congestion information make allocation more fair and efficient?

4.1 Speed of Fast Start

Fast Start allows connections in RackCC to ramp up immediately to their fair share of the network. We use a simple model to quantify the the relative speed up between RackCC's Fast Start and TCP's slow start mechanisms. We assume that RackCC computes and advertises each flow's fair share after a fixed amount of time. This is as opposed to TCP slow start, which doubles its window every RTT from initial window of 6 KB until the rate matches subflow's fair share. Each subflow's rate should be stable around its fair share.

The benefit of Fast Start depends on many factors including the traffic pattern, size of the new subflow, and the latency to determine the fair share of the new subflow. We ran this experiment with many configurations to evaluate how these parameters affect the speedup of RackCC versus traditional TCP. We draw JumboFlow's existing bandwidth throughput from an exponential distribution from previous measurements [6]. The results are plotted in Figure 2. As expected, the size of the new flow matters, and RackCC provides a better speedup for short flows because it removes TCP's need to reprobe the path. For long flows, the latency of TCP's slow start is amortized. The latency for RackCC to determine fair share also matters. If server takes 1 RTT to compute the fair share, we achieve a 3.2× speedup for short flows. If server takes 3 RTTs to compute the fair share, we only provide a 1.3× speedup for the same flows. Finally, when the number of subflows in the JumboFlow is larger, the target throughput for the new subflow is smaller and thus RackCC has a smaller speedup because the it takes fewer RTTs for TCP to converge to the steady state.

4.2 Effect of Cooperation

In RackCC, connections cooperate with other connections in the same rack in order to achieve more accurate, fairer congestion detection and handling. To quantify the effect of this inter-flow aggregation, we ran an event-driven, packet-level simulator. Switches in our simulator use standard drop-tail queues with a per-port buffers based on the bandwidth-delay product of the network. The network is a three level FatTree with a total network latency of around 60 us and an aggressive min-RTO of 200 us. In this network, we implemented a JumboFlow and its congestion logic. As a comparison, we also implemented standard DCTCP [4]. In both cases the ECN threshold is set to be ⅙ of the total queue length.

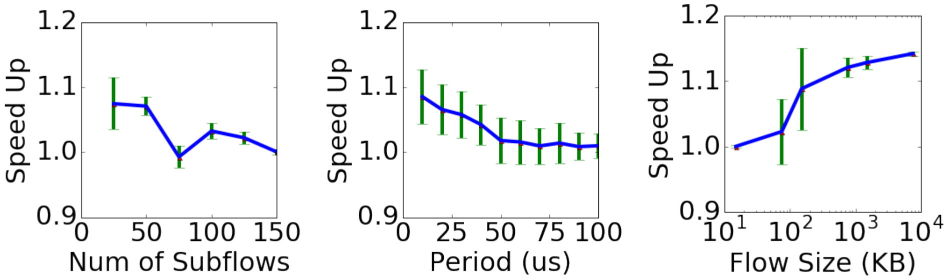
Our small test had five servers in the same rack send equal amounts of traffic to five other servers in a different rack. All the flows are within a single JumboFlow. Figure 3 shows the speed up of the tail flow completion time for RackCC comparing to DCTCP.

Because every DCTCP flow tries to estimate congestion using its own ACKs, every flow has a different estimation and thus a different speed. This causes tail latency to suffer. In RackCC, on the other hand, all the subflows use a single, shared ECN rate. This reduces the tail completion time of the set of flows especially in the case of short flows. The amount of speed up varies across different configurations. When we test one parameter, we keep other parameters at their default values (number of subflows = 25, ECN aggregation period = 10 us, per-subflow flow size = 150 KB). Figure 3a shows that when the total number of subflows increases, the speed up decreases. Figure 3b shows that speed up decreases when as the ECN aggregation period decreases since the ECN information becomes increasingly stale. Figure 3c shows that when flow sizes increase, speedup increases and then plateaus.

5. DISCUSSION

What we have described thus far is a system that replaces TCP's congestion control protocol. Flow control and error detection can be layered on top of this system exactly as they are in traditional TCP except that flow control is rate based, rather than window based. Reliability can be incorporated independently of the switches as well, as every packet still has a sequence number. Thus, dup-acks, timeouts and similar mechanisms are still an effective way to detect drops.

Coexistence with UDP RackCC applies congestion control to all connections, including UDP flows. Note that this means a UDP flow will need to register and unregister themselves



(a) (b) (c)

Figure 3: Job completion time speed up by sharing ECN rate across subflows.

with their ToR and communication partner. However, other than that, UDP retains its properties and is therefore still suitable for traffic that does not require reliable transmission, ordering, etc.

Querying ToR State The fast convergence of the RackCC relies on fast responses from switches. JumboFlow statistics can be queried in-band via any customizable protocol, e.g. SNMP. Throughput can already be estimated using high-resolution switch counters. P4 [7] can be used to specify a hardware pipeline in the switch to calculate the ECN rate and registration values. In addition, P4 can specify a customized packet header to transmit ToR state to servers.

Adaptive Load Balancing Exposing JumboFlow congestion levels to end hosts has other potential benefits. The most obvious benefit is that RackCC can be an easy way to implement clever adaptive load balancing algorithm by picking a particular JumboFlow to join given the presence of multiple redundant paths between ToRs. We can emulate CONGA [3] by always choosing the least congested JumboFlow.

6. RELATED WORK

Data center congestion control is a well-studied topic. Currently, TCP [11] and its variants dominate both the data center and wide area network. Of particular note are recent proposals like DCTCP [4] and MPTCP [18], which seek to address longstanding issues with TCP. DCTCP is a TCP variant that leverages Explicit Congestion Notification(ECN) to reduce buffer usage and speed up flow completion. MPTCP shards data to different paths to better leverage multi-path networks. This necessarily involves incorporating domain-specific knowledge about path selection. In all of these TCP variants, as well as protocols like QCN [2], each flow probes the network and responds to congestion independently. RackCC, on the other hand, shares congestion information between different servers so they may cooperatively arrive at a fairer, more performant rate. In this, RackCC is similar to earlier efforts to share congestion information across different connections within a server [5].

Protocols like XCP [13] and RCP [8] shift congestion control to the network itself. In these proposals, intermediate switches adaptively compute the congestion window/sending rate based on measured rates and observed queuing. In essence, these protocols use the combined information from all flows passing through the current switch to provides fast convergence and fast flow completion times. Though they are effective, they require complex hardware modifications. One way to view our work is as an approach to provide similar properties while keeping most of the complex calculations at the edge. In fact, in RackCC, ToRs only need to provide simple packet/ECN counters and the ability to perform simple arithmetic upon demand.

EyeQ [12] is related in that their sender and receiver rate negotiation proceeds in a similar way to RackCC. However, EyeQ is designed with a different goal in mind: performance isolation. Thus, their negotiated rates are intended to emulate a reserved tunnel on top of which applications will still run TCP. In addition, they ignore congestion in the middle of the network and assume the only bottlenecks are the access links.

Finally, centralized approaches like Fastpass [14] explore the possibility of using centralized controllers to explicitly schedule packet timing. Unfortunately, these proposals do not scale to large data centers due to the sheer size of the network and the burstiness of data center traffic.

7. CONCLUSION

The rack-based architecture of modern data centers provides a potential opportunity for a better way to handle congestion. Indeed, for some traffic patterns (e.g., those with high churn, abundant connections, and locality), today's congestion control algorithms are a poor fit.

In this paper, we instead argue for exploring a different area of the design space: rack-level congestion control. In other words, we seek to facilitate cooperation between disparate connections in a topology-aware fashion. We present a preliminary design based on this design principle. Our system, RackCC, tunnels all connections through rack-to-rack JumboFlows and does so with only a few extra counters at the ToR switch.

8. REFERENCES

[1] SPDY, whitepaper. http://www.chromium.org/spdy/spdy-whitepaper.

[2] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, R. Pan, B. Prabhakar, and M. Seaman. Data center transport mechanisms: Congestion control theory and IEEE standardization. In *Allerton 2008*.

[3] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *SIGCOMM 2014*.

[4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM 2010*.

[5] H. Balakrishnan, H. Rahul, and S. Sesha. An Integrated Congestion Management Architecture for Internet Hosts. In *SIGCOMM 1999*.

[6] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[7] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 2014.

[8] N. Dukkipati. *Rate Control Protocol (RCP): Congestion Control to Make Flows Complete Quickly*. PhD thesis.

[9] M. Ghobadi, R. Mahajan, A. Phanishayee, N. Devanur, J. Kulkarni, G. Ranade, P.-A. Blanche, H. Rastegarfar, M. Glick, and D. Kilper. ProjecToR: Agile Reconfigurable Data Center Interconnect. In *SIGCOMM*, 2016.

[10] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS OSR 2008*.

[11] V. Jacobson. Congestion Avoidance and Control. *SIGCOMM CCR 1988*.

[12] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI 2013*.

[13] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-delay Product Networks. In *SIGCOMM 2002*.

[14] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized "Zero-queue" Datacenter Network. In *SIGCOMM 2014*.

[15] V. Rao and E. Smith. Facebook's new front-end server design delivers on performance without sucking up power. https://code.facebook.com/posts/1711485769063510/facebook-s-new-front-end-server-design-delivers-on-performance-without-sucking-up-power/.

[16] A. Roy, H. Zeng, J. Bagga, G. M. Porter, and A. C. Snoeren. Inside the Social Network's (Datacenter) Network. In *SIGCOMM 2015*.

[17] Verma, Abhishek and Pedrosa, Luis and Korupolu, Madhukar and Oppenheimer, David and Tune, Eric and Wilkes, John. Large-scale cluster management at Google with Borg. In *Eurosys 2015*.

[18] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *NSDI 2011*.