XCo: Explicit Coordination to Prevent Network Fabric Congestion in Cloud Computing Cluster Platforms

Vijay Shankar Rajanna, Smit Shah, Anand Jahagirdar, Christopher Lemoine, and Kartik Gopalan
Computer Science, Binghamton University (State University of New York)
Binghamton, NY, 13902-6000, USA
Contact: kartik@binghamton.edu

ABSTRACT

Large cluster-based cloud computing platforms increasingly use commodity Ethernet technologies, such as Gigabit Ethernet, 10GigE, and Fibre Channel over Ethernet (FCoE), for intra-cluster communication. Traffic congestion can become a performance concern in the Ethernet due to consolidation of data, storage, and control traffic over a common layer-2 fabric, as well as consolidation of multiple virtual machines (VMs) over less physical hardware. Even as networking vendors race to develop switch-level hardware support for congestion management, we make the case that virtualization has opened up a complementary set of opportunities to reduce or even eliminate network congestion in cloud computing clusters. We present the design, implementation, and evaluation of a system called XCo, that performs *explicit coordination* of network transmissions over a shared Ethernet fabric to proactively prevent network congestion. XCo is a software-only distributed solution executing only in the end-nodes. A central controller uses explicit permissions to temporally separate (at millisecond granularity) the transmissions from competing senders through congested links. XCo is fully transparent to applications, presently deployable, and independent of any switch-level hardware support. We present a detailed evaluation of our XCo prototype across a number of network congestion scenarios, and demonstrate that XCo significantly improves network performance during periods of congestion.

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Communications Management–Network Communication; D.4.7 [**Organization and Design**]: Distributed System

General Terms

Design, Experimentation, Performance

Keywords

Congestion, Ethernet, Virtualization

1. INTRODUCTION

Cloud computing infrastructures consist of large data centers clusters that increasingly use commodity servers and networking hardware, such as Gigabit Ethernet. Hardware, administration, and energy costs are driving cloud operators to use virtualization technology to consolidate thousands of virtual machines (VMs) on shared hardware platforms. Most VMs host service-oriented applications that are inherently communication intensive. Consequently, congestion in the cluster's network fabric is becoming a major concern, even in networks with multi-Gigabit switching capacity.

A number of converging factors contribute to this congestion, such as storage access over network using iSCSI [18] or Fibre Channel over Ethernet (FCoE) [17], data-intensive applications such as streaming media and data mining, and aggressive consolidation of communication-intensive VMs onto a limited number of servers and switches. Additionally, the network fabric carries high volume control traffic generated by the virtualization layer for activities such as live VM migration, VM checkpointing, and backups. Commodity Ethernet hardware is cheap, easy to install and manage, and can be shared by a wide range of network services and protocols. However, this commoditization often comes at a price, namely higher latency and smaller/lower-performance packet buffers. Consequently, switch buffers can become easily overwhelmed by high-throughput traffic that can be bursty and synchronized, leading to significant packet losses.

A well-known example of congestion in data center Ethernet is the TCP throughput collapse problem, also known as Incast [26, 35, 28, 9], that is experienced by barrier-synchronized traffic, such as synchronous reads in networked storage. Incast arises from synchronized packet losses suffered by multiple TCP senders at intermediate switch buffers leading to lock-step timeouts and significant under-utilization of network capacity. Another source of network congestion could be the presence of non-TCP traffic, such as UDP, or traffic that does not self-regulate in response to network congestion. For example, Facebook engineers rewrote `memcached` [31] to use *UDP traffic* to enable application-level flow-control. Other types of traffic may be responsive to congestion but not TCP-friendly [10], such as streaming media, voice/video over IP, and peer-to-peer traffic. Furthermore, even in the absence of significant UDP traffic, large number of short-lived TCP connections (mice) have been found to be common in data centers [13]. Simultaneous bursts of short TCP connections can generate a congestion effect similar to UDP [6] since they collectively transmit heavy bursts of packets during the TCP slow start phase.

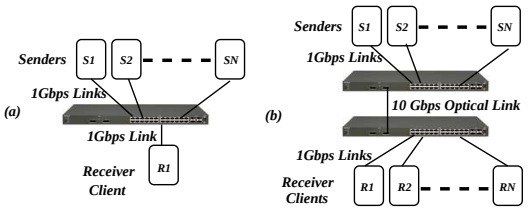

Figure 1: Experimental setups: Multiple senders transmit to (a) one receiver via 1Gbps link, (b) different receivers via 10Gbps uplink.

Essentially, the root cause of all congestion is the transient overload of buffering capacity within a switch. Hardware and software mechanisms to control congestion in commodity Ethernet switches are hard to deploy at scale [19, 16]. Ethernet flow control in 802.3x [11] allows an overloaded downstream port to request a temporary pause of all traffic from the upstream port. While useful in low-end edge switches, this feature is counter-productive in backbone switches due to head-of-line blocking effect. Thus administrators are often reluctant to enable it for fear of slowing down switch forwarding performance. The current industry practice is to simply throw more hardware at the problem by adding higher capacity network switches, multi-port network cards, and physically separate layer-2 networks for data and control traffic. However additional hardware merely increases network cost and complexity without addressing the root cause of the problem.

This paper makes the case for *explicit coordination* of network transmission activities among virtual machines (VMs) in the data center Ethernet to proactively prevent network congestion. We argue that virtualization has opened up new opportunities for explicit coordination that are simple, effective, currently feasible, and independent of switch-level hardware support. We show that explicit coordination can be implemented transparently without modifying any applications, standard protocols, network switches, or VMs. Our solution, called XCo, co-ordinates the network transmissions from multiple VMs so to avoid throughput collapse, while simultaneously increasing network utilization. We present experimental evidence via a proof-of-concept implementation of XCo that demonstrates significant gains in switched Ethernet performance during periods of network congestion.

We begin by experimentally demonstrating congestion-induced performance problems in a Gigabit Ethernet fabric under a number of different scenarios. Next we present the design and implementation of XCo and demonstrate that it can prevent throughput collapse and significantly improve network utilization. Finally we discuss related research and conclude with an overview of challenges and opportunities in a full-fledged XCo-based solution.

2. IMPACT OF ETHERNET CONGESTION

This section motivates the need for explicit coordination by experimentally demonstrating the performance impact of network congestion. Consider two experimental setups shown in Figure 1. End nodes running Xen [3] VMs are connected via a layer-2 switched Ethernet consisting of two Nortel 4526-GTX switches each having 24 1000Base-T ports (for end host connectivity) and two XFP slots with 10Gbps optical transceiver modules (for uplink between switches). Hosts run Xen 3.3.1/Linux 2.6.29.2. Although both setups

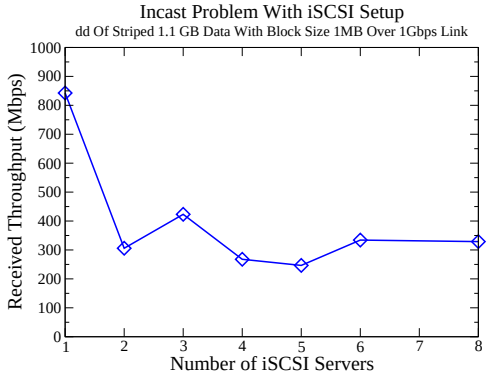

Figure 2: Incast problem with iSCSI setup.

are somewhat simple, they serve to demonstrate the basic traffic contention problem in larger switched Ethernet.

2.1 TCP Throughput Collapse: Incast

We first demonstrate the problem of TCP throughput collapse, also known as the Incast problem, that was documented earlier in [26, 35, 28, 9]. This problem is observed in high-bandwidth low-delay networks where multiple servers send barrier-synchronized traffic to a common client.

To demonstrate this problem, we set up an iSCSI storage network with the topology shown in Figure 1(a). We stripe the data using RAID-0 configuration across N storage servers (iSCSI targets) $S1...SN$. The storage client $R1$, an iSCSI initiator, reads 1GB of this striped data using the `dd` command using a block unit size of 1MB. The network interfaces at both the client and the server nodes use jumbo frames of size 9KB. Figure 2 shows that, as the number of iSCSI targets is increased, the throughput reported by `dd` drops steeply from 842Mbps to 305Mbps. Despite the fact that more iSCSI servers means that each read of a 1MB block is serviced in parallel by multiple servers, the near simultaneous response from all servers overwhelms the network switch buffer, leading to dropped packets and lower read throughput than expected.

When a packet is dropped, TCP waits for at least RTO_{min} time before retransmitting the packet. This in turn delays the client's request for next block of data. The ratio of TCP's RTO_{min} (typically 200ms) to the time taken to transfer the data blocks is high. Consequently, synchronized packet losses and timeouts result in large network idle times leading to TCP throughput collapse. The iSCSI application demonstrated above is one instance of a more general class of cluster-based applications which generate barrier-synchronized traffic such as parallel I/O in cluster file systems [4, 26, 33], parallel back-end responses to search queries in `memcached` clusters [23], or in parallel query processing in distributed databases.

2.2 Throughput Collapse with Non-TCP Flows

We now show that even in the absence of synchronized traffic, the presence of traffic that is not responsive to congestion can lead to throughput collapse in Gigabit Ethernet. Figure 1(a) shows a setup where five different hosts send a mix of TCP and UDP network traffic to a common receiver on another host. Each sender uses the `netperf` [27] benchmark to generate either TCP or UDP traffic to a `netperf` server on the receiver. Figure 3 shows the total (aggregate)

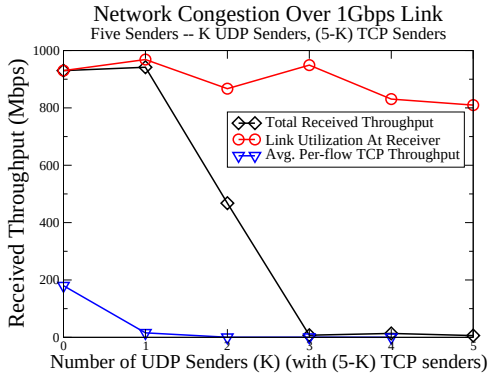

Figure 3: Collapse at 1Gbps link in Fig 1(a).

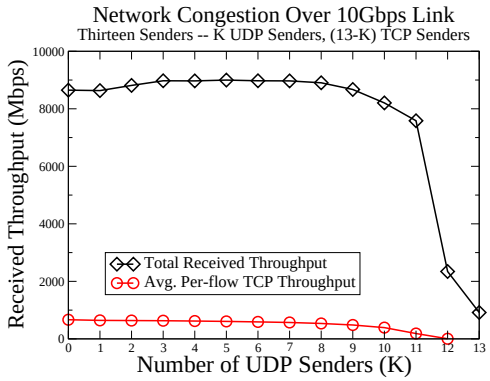

Figure 4: Collapse at 10Gbps link in Fig 1(b).

received throughput as the number of UDP senders is increased and, correspondingly, the number of TCP senders is decreased. All five senders transmit without any coordination. When all five senders transmit TCP traffic, the total throughput is more than 900Mbps. As we increase the number of UDP senders in the mix, we see an immediate drop in the total received throughput.

The figure also shows that the link utilization at the receiver's 1Gbps link (measured using `tcpdump`) varies between 800Mbps to 1Gbps. Since the MTU size is 1.5KB, but the `netperf` UDP senders transmit 64KB application-level messages, dropping even a single 1.5KB packet leads the receiver to discard the entire 64KB message. Thus, in spite of a high link utilization, the application-level throughput suffers.

Finally, Figure 3 plots the average per-flow TCP throughput, i.e. the total `netperf` TCP throughput divided by number of TCP flows. Once UDP senders start transmitting, the average per-flow TCP throughput drops to a negligible share because the TCP flows reduce their sending rate in response to packet losses, but UDP does not. This problem is precisely why TCP was designed. However, not all network traffic in the cluster may be congestion responsive or TCP-friendly and even a moderate amount of non-TCP traffic can significantly lower the throughput of TCP flows.

To demonstrate further, Figure 1(b) shows another experimental setup where 13 senders on different hosts send a mix of TCP and UDP `netperf` traffic to 13 receivers, again on separate hosts. We connect each sender via a 1Gbps link to one Nortel 4526-GTX switch. Similarly, we connect each receiver to another Nortel switch. The two switches are connected to each other by a 10GigE optical link via their XFP transceiver modules. The plots in Figure 4 shows the total

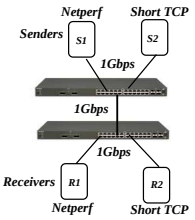

Figure 5: Topology for short TCP flow experiment.

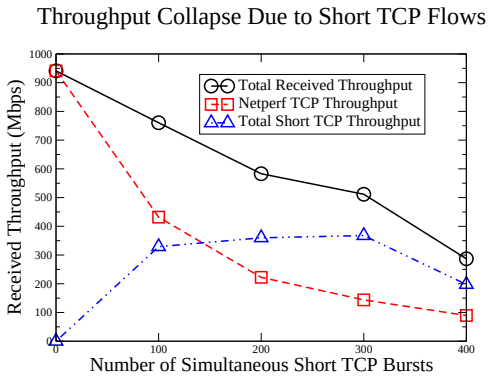

Figure 6: Throughput collapse for a long-lived TCP flow due to multiple short-lived TCP flows.

received throughput across all receivers via the 10Gbps link. As we increase the number of UDP senders (and correspondingly decrease the number of TCP senders), there is a steep drop in the total received throughput due to congestion at the first switch. As with the previous experiment, we again observe that both the application-level throughput and the average per-flow TCP throughput collapses with increasing UDP senders. *This experiment shows that simply replacing 1Gbps hardware (switches, links, and network cards) with their 10Gbps counterparts will not solve the network congestion problem.* If anything, it could worsen the problem since a single end-host, possibly running multiple VMs, will be more likely to quickly saturate a 10Gbps pipe originating from its NIC.

2.3 Throughput Collapse with Short TCP Flows

We now show that throughput collapse is possible even in clusters that primarily carry TCP traffic. It has been shown in [21] that numerous concurrent short-lived TCP flows, that are common in cloud computing platforms [13], have similar effect on network congestion as UDP traffic does, leading to low link utilization and throughput collapse for competing long-lived TCP flows. Each short-lived TCP flow sends a burst of data during its slow-start phase, after which it terminates without ever reducing its congestion window.

To reproduce this behavior, we developed a traffic generator, which we call `shorttcp`, that creates K parallel sequences of consecutive short-lived TCP flows between two nodes. After connection establishment, each TCP flow transmits 30KB of data and terminates, followed by another short TCP connection and so on.

For this experiment, we set up a network topology as shown in Figure 5. Sender $S1$ sends a long-lived `netperf` TCP flow to receiver $R1$. Simultaneously, sender $S2$ sends short-lived `shorttcp` traffic to receiver $R2$. The `netperf` and `shorttcp` traffic flows contend for the 1Gbps common uplink between the two switches. Figure 6 shows the throughput

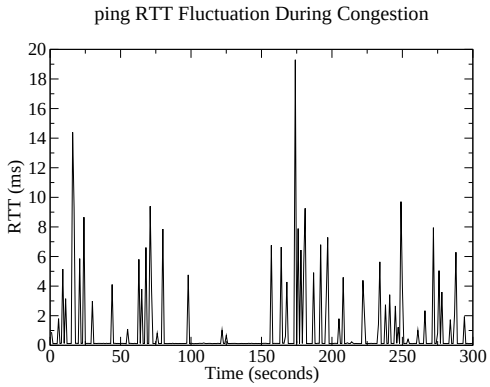

Figure 7: `ping` RTT fluctuation during congestion.

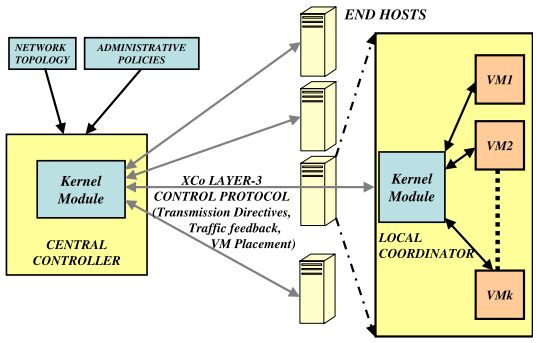

Figure 8: High-level architecture of XCo.

observed for the long-lived `netperf` flow as the number of simultaneous sequences of short-lived TCP flows (K) is varied from 0 to 400. As K increases, `netperf` throughput drops from 940Mbps to 89Mbps. Correspondingly, the total received throughput for all flows collapses from 940Mbps to 280Mbps, demonstrating the throughput collapse effect.

2.4 Impact of Congestion on Response Times

Here we demonstrate that Ethernet congestion could lead to significant variations in network response times, which is detrimental to latency-sensitive applications such as E-commerce and financial transactions. We conducted a simple experiment over the topology shown in Figure 1(b) by measuring the round-trip time (RTT) reported by the `ping` command (ICMP echo/response) in the absence and presence of background network traffic. Without background network traffic, the `ping` command reported an almost constant RTT of $86\mu s$. When the 10Gbps optical link is saturated with background network traffic, consisting of a mix of TCP and UDP `netperf` traffic between different sender and receiver nodes, the RTT reported by `ping` fluctuates heavily between $100\mu s$ and 20 milliseconds, as shown in Figure 7. While this result is somewhat expected, the experiment underscores the fact that without some form of transmission coordination between end-systems, the RTT could experience significant fluctuations during times of congestion.

3. XCO: DESIGN AND IMPLEMENTATION

This section presents the design and implementation of the XCo framework for preventing Ethernet congestion. Although presented in the context of Xen [3] virtualization platform, the fundamental concepts presented here are applicable across different virtualization technologies.

3.1 Overview of XCo Framework

XCo works by preventing multiple end-hosts from sending too much data at the same time. The cluster administrator may not have complete control over the type of traffic VMs generate, such as in a subscriber-based cloud computing model where different external users may own individual VMs. The cluster administrator also may not have the freedom to modify the application or operating system image within each VM. *However, the cluster administrator does have control over a privileged VM (such as Domain 0 in Xen), which can intercept, monitor, and control the traffic being transmitted from all other VMs in a host machine.*

Figure 8 shows one *central controller* that resides in the same switched network as the other nodes. The central controller could be mirrored by a hot-standby node for fault-tolerance. The central controller takes as input (i) the switch interconnection topology and link capacities, (ii) the location of VMs on physical nodes, and (iii) the current traffic matrix of the network. Whenever the central controller detects congestion buildup at any link, it computes and generates transmission directives (or explicit permissions to send) to local coordinators at each end-host that is contributing to the congestion.

Each *local coordinator* intercepts and regulates the outgoing traffic aggregates from all VMs within the end-host and provides traffic feedback to the central controller. For discussions in this paper, a traffic aggregate is defined at the granularity of a VM-to-VM flow, or a *V2V flow*, i.e. traffic from one VM to another VM. Our prototype presently identifies V2V flows using source-VM and destination-VM IP address pairs.

At a high level, V2V flows traverse bottleneck links in the network and local coordinators regulate the V2V flows that contribute to congestion. The specific regulation pattern is dictated by *transmission directives*, or explicit instructions, from the central controller to each local coordinator every few milliseconds. Transmission directives could take many forms. For example, a transmission directive could take the form of *explicit timeslice scheduling*, such as "which V2V flow transmits packets when and for how long." Or a transmission directive could be *explicit rate limiting*, such as "at what rate should a V2V flow transmit for the next N milliseconds." Or the directives could be a combination of both or any other form suited to the performance objectives.

Our rationale for using a central controller to coordinate transmissions among potentially thousands of nodes is that data center clusters are already tightly coupled with extensive central monitoring and control. For example, centralized control has played a key role in recent large-scale deployments of storage [12] and data processing systems [9] in the scale of tens of thousands of machines. Similarly, Hedera [2] and Ethane [6] are centralized flow-scheduling systems for data center and enterprise Ethernets. Nevertheless, the framework proposed in this paper could likely be adapted to a distributed implementation for situations where requirements preclude centralized control.

In theory, if the central controller can carefully coordinate the transmission activities of all V2V flows across the data center, then one could precisely control the extent of network load at each link in the switched network. Of course,

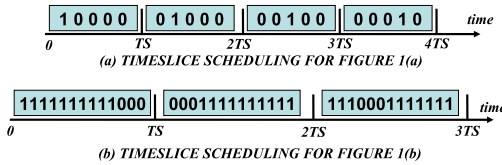

Figure 9: Timeslice scheduling for Figure 1 setup. Value of 1 indicates that the corresponding sender is allowed to transmit during the timeslice.

there are a number of practical challenges to implementing such a network-wide transmission control. In this paper, we investigate these challenges through one form of central coordination, namely *timeslice scheduling*.

3.2 Timeslice Scheduling

The central controller temporally divides network transmission into timeslices and explicitly decides which V2V flows transmit in each timeslice. *The goal is to prevent excessive concurrent transmissions that can potentially cause congestion in some part of the switched network, while permitting sufficient network activity to achieve high utilization.* When a scheduling event (defined in Section 3.2.1) occurs, the central controller computes and sends transmission directives to one or more local coordinators indicating which V2V flows are eligible to transmit during that timeslice. The timeslice granularity is small, in the order of 1ms to 10ms.

For example, consider the setup shown in Figure 1(a) where the common receiver is susceptible to throughput collapse. In this case, when the central controller detects that the link utilization is close to its full capacity, it permits only one of the competing V2V flows to transmit at a time. Thus the central controller sends the transmission directive shown in Figure 9(a) every 10ms, permitting only one V2V flow to proceed in each slot.

Similarly, consider the setup shown in Figure 1(b) where the output port leading to the 10Gbps uplink is congested. In this case, it would be inefficient to allow only one V2V flow in each timeslice since each end-host is capable of transmitting at a maximum of only 1Gbps. Hence the central controller permits up to 10 V2V flows to transmit simultaneously in each timeslice by sending the sequence of transmission directives shown in Figure 9(b).

Distributed Work Conservation: Some nodes may finish with their timeslice early, leaving unused capacity at the bottleneck link. To prevent network under-utilization in such cases, we designed timeslice scheduling to be *work-conserving* [36]. The local coordinator gives up its unused timeslice back to the central controller. The central controller then permits another node to transmit. To avoid giving up a timeslice too quickly, the local coordinator introduces a small hysteresis delay (few tens of microseconds) before returning the unused timeslice, hoping that more packets might arrive during the delay.

3.2.1 Timeslice Scheduling Algorithm

We now present an algorithm for timeslice scheduling that generalizes the above approach for an arbitrary Ethernet topology and dynamic communication pattern among nodes.

Notations and Assumptions: Consider a network with a set N of physical nodes (end-hosts). Each node can host one or more VMs. Assume that the central controller knows the interconnection topology of all the layer-2 switches in the network, which could be arbitrary. There are L links in the network topology. A link from a i to j is represented by an ordered pair (i, j), where i and j could be either switches or end-hosts. Conversely, the link in the reverse direction from j to i is represented by (j, i), although physically a single bidirectional cable might connect the two nodes. Assume that the central controller has knowledge of the subset of links ST which constitute the spanning tree among switches at any point in time. This information can be queried from modern managed switches via their management interfaces. The transmission capacity of link (i, j) is represented by C_{ij}. For ease of exposition, the capacity of the outgoing link C_{xj} from an end-host x to switch j is represented by C_x (assuming there is only one outgoing link per end-host). Given the spanning tree ST, we pre-compute the path P_{sd} from every source node s to every other destination node d in the network.

$$P_{sd} = \{\ (i,j) \mid (i,j) \text{ lies in the path from } s \text{ to } d \text{ in the spanning tree } ST\ \}$$

Backlog Group: B_x of a node x is the set of destination nodes for whom x has backlogged V2V flows, i.e. V2V flows that have packets in their queues ready to be transmitted. A *backlogged node* x is a node with a non-empty backlog group B_x. Each node x updates and reports its backlog group B_x to the central controller whenever any one of its V2V flows changes from a non-backlogged to backlogged state.

Transmission Directive: Upon every scheduling event, the central controller computes and sends transmission directives to one or more nodes in the network. For timeslice scheduling, a *transmission directive* to a source node s is defined as the single destination node D_s to which s is allowed to send during the next timeslice at full outgoing link capacity C_s. Node s then allows all V2V flows having destination in D_s to transmit. If $D_s = null$, then s is not allowed to transmit in the next timeslice.

A **scheduling event** is one of the following occurrences:

1. The timeslice of a currently scheduled node expires (via a timer expiry at central controller).
2. A node s voluntarily gives up its unused timeslice for work-conservation through an explicit message to the central controller. This event indicates that s does not have any more packets to send to the destination D_s. Both node s and the central controller remove destination D_s from the backlog group B_s.
3. A previously non-backlogged node s (with $B_s = \emptyset$) becomes backlogged ($B_s \neq \emptyset$) and sends its new (non-empty) backlog group B_s to the central controller.

A **contention group** β_{ij} is the set of nodes with backlogged traffic whose path includes link (i, j).

$$\beta_{ij} = \{\ s \mid \exists d \text{ where } d \in B_s \text{ and } (i,j) \in P_{sd}\ \}$$

The contention groups β_{ij} are updated by the central controller every time a new backlog set B_s is reported by a node. The **active contention group** α_{ij} is the set of nodes actively sending traffic through link (i, j) at any instant. α_{ij} is a subset of β_{ij}

$$\alpha_{ij} = \{\ s \mid D_s \neq null \text{ and } (i,j) \in P_{sD_s}\ \}$$

Feasibility Condition: A link (i, j) will not experience congestion if the sum of all traffic allowed through the link

Algorithm 1 Algorithm to compute transmission directive upon a scheduling event.

```
1: Input: (a) Current spanning tree ST
2:       (b) Maximum link capacity C_ij ∀(i.j) ∈ ST
3:       (c) Pre-computed paths P_xy ∀ nodes x, y
4:       (d) Current backlog group B_x ∀ nodes x
5:       (e) Current contention group β_ij ∀ links (i, j)
6:       (f) Current active contention group α_ij ∀ links (i, j)
7:       (g) Last transmission directive D_x^old ∀ nodes x
8:       (h) Type of scheduling event
9:       (i) Node t which triggered the scheduling event
10: Output: Next transmission directive D_x for each node
    x affected by the scheduling event
11:
12: A := ∅ /*set of nodes affected by scheduling event*/
13: for each node d ∈ B_t do
14:   for each link (i, j) ∈ P_td do
15:     A := A ∪ β_ij
16:   end for
17: end for
18: if D_t^old ≠ null then
19:   for each link (i, j) ∈ P_{tD_t^old} do
20:     α_ij := α_ij − {t}
21:   end for
22:   if (scheduling event = work conservation) then
23:     B_t := B_t − {D_t^old} /*t has no backlog to D_t^old*/
24:   end if
25: end if
26: N := ∅ /*set of nodes with new schedule*/
27: while A ≠ ∅ do
28:   x := some node in A
29:   D_x := null
30:   for each node d ∈ B_x do
31:     for each link (i, j) ∈ P_xd do
32:       /*Check feasibility condition*/
33:       if F({x} ∪ α_ij, C_ij) = false then
34:         Skip to next d in line 30
35:       end if
36:     end for
37:
38:     D_x := d /*d satisfies feasibility at each link*/
39:
40:     if D_x^old ≠ null then
41:       /*x will stop transmitting to D_x^old*/
42:       for each link (i, j) ∈ (P_{xD_x^old} − P_{xD_x}) do
43:         α_ij := α_ij − {x}
44:         A := A ∪ (β_ij − N) /*more nodes affected*/
45:       end for
46:     end if
47:     for each link (i, j) ∈ P_xd do
48:       α_ij := α_ij ∪ {x}
49:     end for
50:     break;
51:   end for
52:   A := A − {x}
53:   if D_x ≠ null then
54:     /*newly scheduled; don't reschedule again*/
55:     N := N ∪ x
56:   end if
57: end while
58: for each x ∈ N do
59:   Send D_x to x
60:   D_x^old := D_x
61: end for
```

does not exceed its capacity C_{ij}. Formally, the feasibility condition $F(\alpha_{ij}, C_{ij})$ is defined as follows:

$$F(\alpha_{ij}, C_{ij}) : \sum_{\forall x \in \alpha_{ij}} C_x \leq C_{ij} \quad (1)$$

Scheduling Algorithm: The central controller uses Algorithm 1 to compute transmission directives upon every scheduling event. The algorithm computes and maintains the affected set A of nodes whose schedule can potentially change due to the scheduling event. The main loop of the algorithm (line 27) iterates over each affected node x, attempting to compute a new transmission directive D_x. For each destination to which x has backlogged traffic, the algorithm walks the path P_{xd}. At each link (i, j), it checks the feasibility condition to test if the link can accommodate an addition load of C_x. The algorithm selects as D_x (line 38) the first destination d which is feasible along the entire path P_{xd} and then moves on to the next affected node.

Preemption of Active Flows: Observe that an affected node x could be already busy transmitting packets to D_x^{old} while the central controller computes a new transmission directive D_x. In Algorithm 1, we choose the policy of pre-empting (stopping) x's current transmission to D_x^{old} in favor of the newly computed destination D_x (lines 41– 45). Otherwise pathological cases can arise where traffic of some backlogged V2V flows might be starved indefinitely.

Also note that pre-empting an active flow could release resource along additional paths in the network. Additional nodes affected by release of resources along new paths are added back into the affected set A to possibly recompute additional transmission directives (line 44). The algorithm thus tries to maximize the network utilization by allowing as many nodes to transmit as possible without triggering congestion at any link.

To avoid pre-empting flows that have just recently been started, lines 15 and 44 can be modified to exclude from A nodes that have been active for less than a threshold duration. Algorithm 1 omits this detail for clarity.

Generating Distinct Successive Schedules: The central controller needs to ensure that distinct transmission schedules are generated across successive scheduling events and that no backlogged traffic is indefinitely starved. An astute reader would note that the algorithm might end up generating the same schedule across consecutive scheduling events ($D_x = D_x^{old}$) if the order of node traversals in lines 28 and 30 remains fixed. Therefore, the algorithm must traverse the nodes in a different order upon every scheduling event. Again, Algorithm 1 omits this detail for clarity.

Fairness: Another observation is that the algorithm does not make any special effort to maintain fairness across flows, especially TCP-style min-max fairness. We expect that coordinated scheduling would be invoked only when the network is approaching a congested state. Algorithm 1 implements a policy that considers preventing congestion more important than maintaining fairness. Alternative scheduling policies could consider flow fairness more important.

Convergence: Finally, despite possible additions to the affected set A in line 44, the algorithm is guaranteed to converge because (a) nodes once considered are excluded from A (line 52), and (b) nodes once successfully scheduled are tracked (line 55) and not included in A again (line 44).

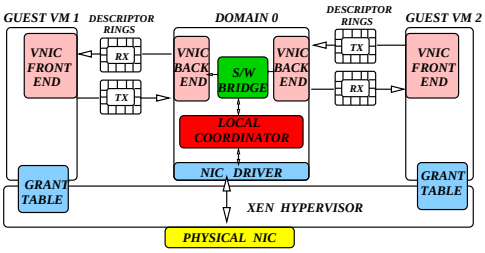

Figure 10: Xen networking subsystem in the end-host and placement of the local coordinator.

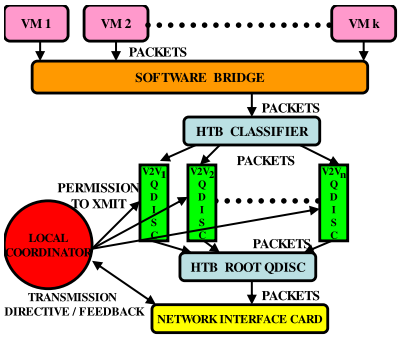

Figure 11: Architecture of the local coordinator.

3.3 Rate Limiting vs. Timeslice Scheduling

The coordination framework in Figure 8 is flexible enough to allow for transmission control strategies other than timeslice scheduling described above. For example, the transmission directives from the central controller could alternatively specify the maximum rate at which each V2V flow should transmit until the next directive. This transmission control strategy of *rate limiting* senders differs from timeslice scheduling in that senders can transmit at any time, but at a throttled rate specified by the central controller. This can allow the central controller to dynamically throttle the transmission rates of each node, depending upon per-link traffic conditions. For comparison, we implemented a simple rate limiting policy in which the rate assigned to each V2V flow is the capacity of the bottleneck link divided by the number of senders through the bottleneck. Our preliminary evaluations in Figures 12, 13, and 14 indicate that rate limiting does improve the received throughput but remains susceptible to throughput collapse with increasing number of senders. A detailed investigation of more sophisticated rate limiting strategies remains part of our future work.

3.4 Implementation Details

Due to requirements of low-overhead and time-sensitive operations, both central controller and local coordinators are implemented as kernel modules in Linux/Xen architecture. We first present a background of the Xen networking subsystem and then describe the implementation issues in some of the major individual components of XCo.

Xen Network Subsystem Background: As shown in Figure 10, Xen exports virtualized views of network devices to each VM, as opposed to real physical network cards. The actual network drivers that interact with the real network card execute within Domain 0 – a privileged domain that can directly access all hardware in the system. The privileged Domain 0 and the unprivileged VMs communicate by means of a split network-driver architecture. Domain 0 hosts the backend of the split network driver, called *netback*, and the VM hosts the frontend, called *netfront*. All netbacks attach to a software bridge in Domain 0, which multiplexes and demultiplexes packets from/to the VMs.

Local Coordinator: Without XCo, all outgoing packets at an end-host would be directly forwarded by the software bridge in Domain 0 to the native driver for the network interface card (NIC). XCo interposes a *local coordinator* module after the software bridge to control transmissions. Figure 11 shows the internal architecture of the local coordinator, which performs transmission control across multiple V2V flows. We first created a custom Linux Traffic Control [22] setup in Domain 0. We use Hierarchical Token Bucket (HTB) queuing discipline (qdisc) as the root queue communicating with the network card. This root queue is configured to contain one Priority (PRIO) qdisc for each V2V flow originating from the end-host. The root HTB qdisc has a packet classifier which queues each V2V flow's outgoing packets in their corresponding PRIO qdiscs. The root HTB qdisc dequeues packets from these PRIO qdiscs for transmission over the NIC. We modified the implementation of PRIO qdisc in Domain 0 to add the local coordinator. For low-overhead communication, the local coordinator and the central controller communicate directly using a new layer-3 protocol type for control messages and thus bypass the overhead of socket-based TCP/IP interface. For timeslice scheduling, only the PRIO qdiscs of V2V flows which are permitted in the directive from central controller can transmit. For rate limiting, each PRIO qdisc transmits at or below the rate specified in the directive.

Central Controller: The central controller must respond to scheduling events with low latency. Consequently, we implemented the central controller as a Linux kernel thread that does nothing but wait for scheduling events, make scheduling decisions, and send transmission directives to local coordinators. The kernel thread constantly polls the system's high resolution clock (via `do_gettimeofday()`) to detect the expiration of a node's timeslice with a finer granularity than the default Linux kernel time – `jiffy` (which can be anything from 1ms to 10ms).

Synchronization: In timeslice scheduling it is important for physical nodes to coordinate their transmissions if they share a bottleneck link. Hence each local coordinator synchronizes its kernel-level "network transmission clock" with the central controller whenever it receives a transmission directive. Within a timeslice the local coordinator uses the local high resolution clock (at microsecond granularity) to track local events.

Dynamic Join/Leave: When a physical node joins the network, its local coordinator registers itself with the central controller. Whenever a new V2V flow is started, the local coordinator sets up its PRIO qdisc. Conversely, whenever a node shuts down, the local coordinator informs the central controller which stops scheduling the node's traffic.

4. PERFORMANCE EVALUATION

We now experimentally demonstrate that XCo can reduce the impact of congestion-induced performance problems in each of the problem scenarios outlined in Section 2. The

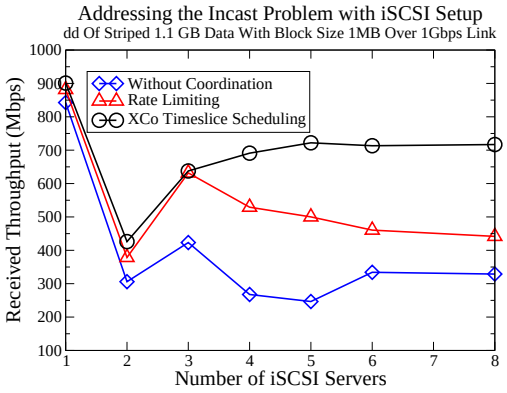

Figure 12: Addressing Incast for iSCSI in Fig.1(a).

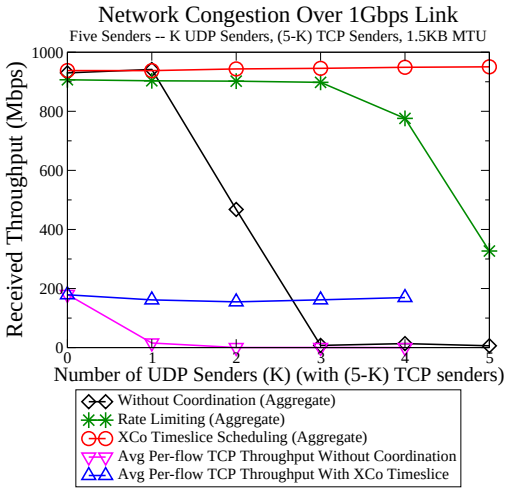

Figure 13: Addressing collapse in Fig 1(a).

experimental setups are the same as in Section 2, except that now we either coordinate the network transmissions using XCo's timeslice scheduling or throttle the transmissions via rate limiting. All timeslices are either 2ms or 10ms.

4.1 Addressing Incast Problem

Recall the experimental iSCSI setup described earlier in Section 2 Figure 1(a) – a storage client (iSCSI initiator) reads 1.1GB of data that is striped using RAID0 configuration over multiple storage servers (iSCSI targets). In Figure 2, we demonstrated the Incast problem which causes a steep drop in the received throughput at the iSCSI client with increasing number of iSCSI targets. Here we show that XCo can address the Incast problem by explicitly (and transparently) coordinating the concurrent transmission activities of multiple senders. Figure 12 shows that timeslice scheduling yields the highest receive throughput. Rate limiting provides a better received throughput than having no coordination at all, but worse than timeslice scheduling. This is because merely reducing the transmission rate does not appear to eliminate transient overload of switch buffers. An initial reduction in observed throughput for smaller number of servers is likely due to inefficiencies in our implementation of distributed work conservation.

4.2 Throughput Collapse with Non-TCP Flows

Now we reconsider the problem of throughput collapse due to non-TCP traffic demonstrated earlier in Figures 3

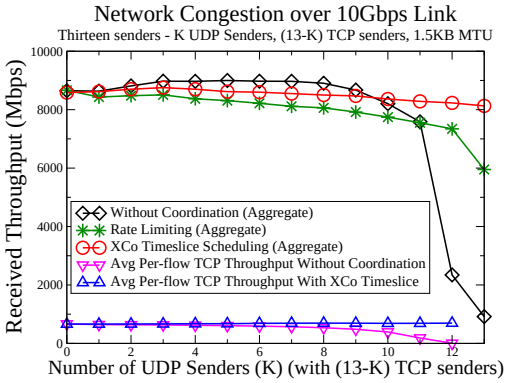

Figure 14: Addressing collapse in Fig 1(b)

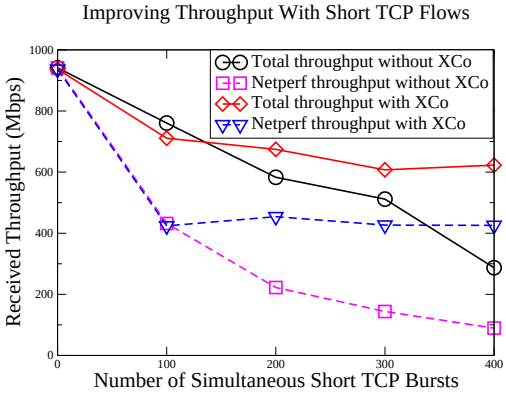

Figure 15: Addressing collapse due to short TCP.

and 4. Recall Figure 1(a) where the common receiver is susceptible to throughput collapse. Figure 13 shows that timeslice scheduling achieves significant improvement in received throughput when using XCo. The received throughput stays close to the maximum 1Gbps even as the number of UDP senders increase. Rate limiting, which throttles each V2V flow to 1/5Gbps, performs better than having no coordination, but worse than timeslice scheduling. Furthermore, the average per-flow TCP throughput remains steady at close to 200Mbps, despite the presence of UDP traffic.

Similarly, recall that in Figure 1(b), the 10Gbps uplink is susceptible to congestion. With timeslice scheduling, the central controller in XCo permits up to 10 V2V flows to transmit simultaneously in each timeslice according to the sequence shown in Figure 9(b). With rate limiting, the central controller throttles each sender to 10/13 Gbps. Figure 14 shows that timeslice scheduling achieves a high aggregate received throughput even as the number of UDP senders increases. As before, rate limiting performs better than no coordination and slightly worse than timeslice scheduling. Finally, TCP senders maintain their throughput share even in the presence of UDP senders.

4.3 Throughput Collapse: Short TCP Flows

We showed in Section 2.3 and Figure 6 that large number of short-lived TCP flows (or mice) can cause unfair throughput collapse for competing long-lived TCP flows (or elephants) besides reducing the total received throughput. In this section, we show that XCo can prevent such a throughput collapse by regulating the network transmissions from individual end systems. We repeat the same experiment as

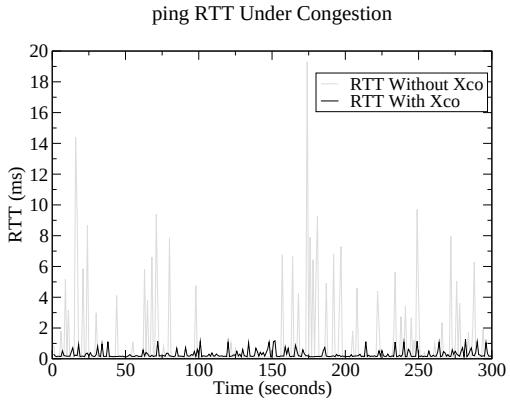

Figure 16: Improving ping RTT under congestion.

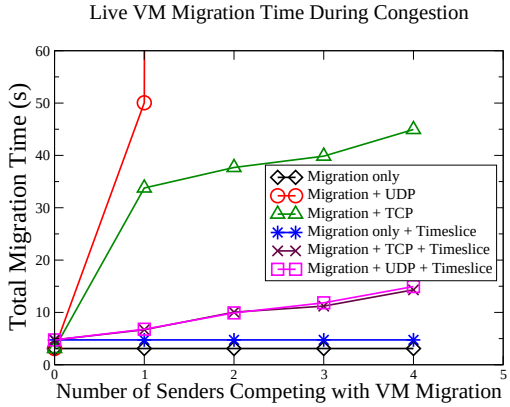

Figure 17: Improving live VM migration time.

in Section 2.3 with competing long-lived `netperf` and short-lived `shorttcp` traffic. Only this time we configure XCo to perform timeslice scheduling among the two senders and two receivers. Figure 15 shows that, with increasing number of `shorttcp` sequences, `netperf` throughput and total received throughput drop steeply without XCo. However, with XCo, `netperf` throughput stabilizes around 430Mbps, slightly less than half the link bandwidth, and the total received throughput increases compared to without XCo.

4.4 Improving Round-trip Response Times

Now we demonstrate how XCo can improve the response times of latency-sensitive applications during times of network congestion. Recall from Section 2.4 that `ping` RTT fluctuated heavily in the setup in Figure 1(b) in the presence of congesting `netperf` traffic on the common 10GigE link. To reduce the fluctuation in `ping` RTT, we applied XCo to coordinate the transmissions of competing large `netperf` senders from different nodes. Since the `ping` traffic is a small flow, the local XCo coordinator at the corresponding node permits the echo/response packets to be transmitted without waiting for transmission directive from the central controller. Figure 16 shows that the resulting `ping` RTT values with XCo now vary between 100μs and 2ms, the maximum value being ten times smaller than without XCo.

4.5 Improving Live VM Migration Time

We now use a simple example of live VM migration [8] to understand the impact of congestion on a virtualized infrastructure and the benefits of using XCo. Again consider

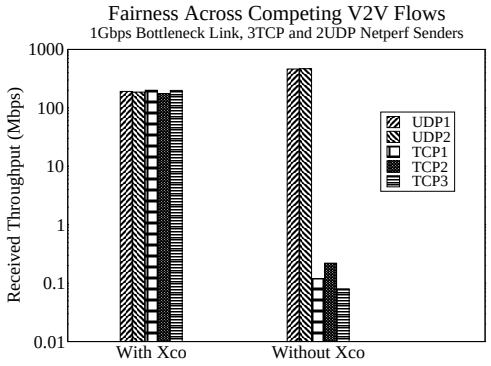

Figure 18: Fairness among competing flows.

the setup in Figure 1(a). We initiate the live migration of a 256MB idle VM from one of the nodes S1 to R1. Simultaneously, we initiate `netperf` traffic from different number of senders S2...S5 to R1. We measure the total time taken to perform live migration as the number of `netperf` senders is increased. In Figure 17, plots labeled "Migration + TCP" and "Migration + UDP" show the total migration time without XCo when all competing `netperf` senders transmit either TCP or UDP traffic respectively. We see that in the presence of even one or more competing `netperf` UDP senders, the total migration time increases exponentially since UDP dominates any available uplink bandwidth. Even when only TCP senders compete, there is still a significant increase in total migration time. Even though all TCP sessions back off during congestion, this does not completely eliminate transient overload of switch buffers. When using XCo, the plots labeled "... + Timeslice" show that the live VM migration time is far less with XCo for both UDP and TCP senders. Although there is still an increase in migration time, the increase is linear and there is almost no difference between the cases of competing TCP and UDP sessions. With competing TCP traffic, the difference between VM migration times with and without XCo can be explained by the observation that link utilization without XCo was less than 500Mbps whereas that with XCo was around 950Mbps. This indicates that, without XCo, TCP senders backoff due to increased packet drops resulting in lower network utilization.

4.6 Fairness Among V2V Flows

To evaluate the extent of fairness enabled by XCo, we conducted an experiment with 2 UDP senders competing with 3 TCP senders that transmit data at maximum possible rate across a 1Gbps bottleneck link. Without XCo, Figure 18 shows that UDP dominates the link bandwidth at the expense of TCP's throughput. With XCo, the TCP flows are able to obtain approximately the same share of bottleneck link bandwidth as the competing UDP flows. Note that fairness enabled by XCo in our current implementation is not max-min fairness, particularly for topologies that are more complex the the simple one evaluated above. This is due to the manner in which the central controller cycles among backlogged flows to avoid starvation and also due to the possibility that a V2V flow might traverse multiple congested links that are subject to different transmission schedules. However, note that if central coordination activates only during congestion, then maintaining high throughput could be more important than guaranteeing fairness.

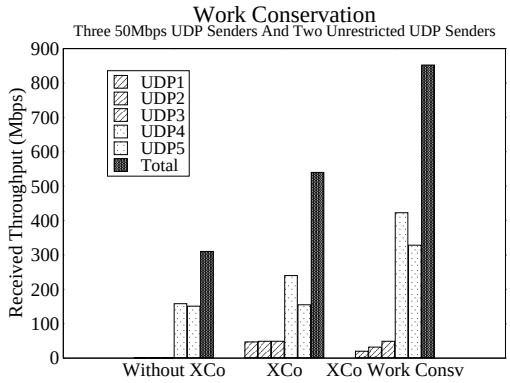

Figure 19: Throughput of five competing UDP flows, with and without work conservation. Three UDP flows transmit at 50Mbps and two UDP flows transmit at the maximum available bandwidth.

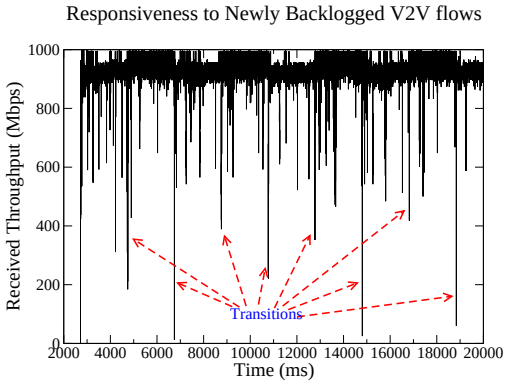

Figure 20: Time series graph showing responsiveness in switching to newly backlogged flows. Two sets of 3 V2V flows alternate every 2 seconds. Each transition between the two sets is less than 10ms.

4.7 Work Conservation

To demonstrate the benefits of work conservation, we conducted an experiment where five UDP senders transmitted to five receivers across a 1Gbps bottleneck link. Three of these flows transmit at 50Mbps and two flows transmit at the maximum available bandwidth. Figure 19 shows the bandwidth achieved without XCo, with XCo but without work conservation, and with XCo and work conservation. When XCo is enabled, total link utilization markedly improves from around 300 Mbps to more than 500Mbps. When work conservation is also enabled, the total link utilization increases to more than 800Mbps, since the spare capacity from the three throttled UDP senders is distributed among the two unrestricted UDP senders. Note that even though complete fairness is not maintained in our implementation of work conservation, received throughput of all flows increases over the case without XCo.

4.8 Responsiveness

We now examine the responsiveness of XCo in scheduling packets from V2V flows that newly arrive or begin transmitting again after a quiet (non-backlogged) period. There are two overheads that affect the responsiveness of XCo to dynamic V2V flows. First is the overhead of creating a classification queue for a new V2V flow for the first time. When a new V2V flow is detected by the local coordinator, our current implementation uses a user-level script in Domain 0, which invokes system commands to classify and queue the packets from the V2V flow based on source-destination IP addresses. We observed that it takes a one-time overhead of 75ms from the arrival of the first packet to the creation of queue for the new V2V flow.

Second is the overhead of re-acquiring the schedule from the central controller when a quiet V2V flow suddenly starts transmitting again. To measure this overhead, we conducted an experiment with six VMs in one node. Three of the six VMs transmit UDP packets for 2 seconds to a common receiver and then stop. Next other three VMs transmit for 2 seconds and then stop. And so on back and forth. Figure 20 shows the time series graph plotting aggregate bandwidth (averaged over 1ms time intervals) at the receiver's link during the transition from one set of VMs to another. The aggregate throughput drops for less than 10ms during the transition before recovering to peak bandwidth. This time includes creating new sender processes in the VMs using scripts, detecting in the local coordinator that the V2V flows are again backlogged, communicating a new backlog set to the controller, and receiving a new transmission directive from the controller. Of the 10ms gap, less than 1ms is expended on communicating the backlog set to the central controller and receiving a new transmission directive. The rest of the gap is due to creating sender processes and communicating initial packets to Domain 0.

5. RELATED WORK

Data Center Ethernet Architectures: The Data Center Bridging Task Group [16] is developing specifications for future hardware QoS support in data center Ethernet fabric through congestion notification, priority based flow control, and enhanced transmission selection. Virtual LANs (VLAN) [15] are extensively used to set up *logical* layer-2 isolation across virtual clusters. However, VLANs do not provide any congestion control or QoS. Hedera [2] is a centralized flow scheduling system for multi-stage switch topologies in data centers that maximizes aggregate network utilization. A central scheduler collects information about larger flows from network switches, computes non-conflicting paths for flows, and manipulates the forwarding tables of the switches to re-route selected flows. Like XCo, Hedera uses a global view of network traffic to determine and alleviate bottlenecks that local schedulers cannot. Unlike XCo, which does not modify the switch forwarding tables, Hedera relies on manipulation of switch forwarding tables to perform multipath routing. Viking [32] and SPAIN [24] construct VLAN-based multiple spanning trees in commodity Ethernet to improve both aggregate network throughput and failure recovery. Like XCo, they rely on a central controller to gather traffic load information and (re)configure the spanning trees. Unlike XCo, which controls traffic at millisecond granularity, Viking performs long-term monitoring and VLAN control (given that VLANs cannot be reconfigured at a high frequencies in most commodity switches), whereas SPAIN only performs offline pre-computation of VLANS. Other projects that advocate the use of a central controller include Ethane [6] for flow scheduling in enterprise networks and the Route Control Platform [5] for routing in wide-area ISP networks. [1, 13, 20, 25] propose alternative forwarding and routing designs to traditional spanning-tree based Ethernet archi-

tectures for data centers. Their objective is to improve fault-tolerance, scalability, and cross-sectional throughput of multi-tiered data center Ethernet topologies through techniques such as improved forwarding logic in switches, multipath routing for load balancing, and reducing layer-2 broadcasts among others. Instead of relying upon sweeping changes to major aspects of Ethernet design or runtime operations (which have their own value but may be impractical in certain settings), XCo chooses a narrow focus of preventing congestion across commodity Ethernet fabric, purely by coordinating the transmission activities among end-points.

TCP Throughput Collapse: Also known as *Incast* problem, TCP throughput collapse was first described in [26] in the context of parallel file systems. The problem was addressed at the application level by limiting the number of servers communicating concurrently with the client and by reducing the advertised TCP receive buffer size. Work in [28] examined a number of TCP improvements to address the Incast problem, concluding that while throughput sometimes improved, none of them substantially prevented TCP throughput collapse. Subsequent work [35, 7, 9] showed that reducing TCP's minimum RTO can help maintain a high throughput, albeit only postponing throughput collapse to a different performance point. Additionally, too small a minimum RTO can lead to spurious timeouts for wide-area network traffic. In addition, none of the TCP-specific solutions to the Incast problem address the case where a large number of short-lived TCP bursts and non-TCP traffic might share the Ethernet fabric, causing severe unfairness to TCP traffic. Timeslice scheduling in XCo can prevent throughput collapse under all scenarios, irrespective of the mix of long-lived TCP, short-lived TCP, and non-TCP traffic sharing the network fabric.

Distributed Rate Limiting (DRL): DRL [29, 34] enforces global rate limits on aggregate traffic used by geographically distributed cloud services, through a set of collaborating rate limiters. XCo also explicitly limits traffic from nodes, although within a data center cluster. Further, XCo differs in its motivation, namely preventing Ethernet congestion, rather than enforcing resource usage limits.

Finally, this work significantly extends our workshop paper [30] with additional algorithm design, implementation effort, and extensive experiments evaluating Incast, short TCP flows, fairness, work-conservation, and responsiveness.

6. FUTURE WORK

While this work highlights the potential of XCo in addressing Ethernet congestion, a number of challenges remain.

Empirical Studies: In this paper, we used simple network setups to study the potential benefits of XCo. More detailed empirical studies are required to evaluate the performance of timeslice scheduling algorithm on complex topologies with multiple bottlenecks, and to evaluate network utilization and starvation avoidance of timeslice scheduling.

Scalability: While there is evidence [6] that central controllers are preferable in managing large clusters, one could possibly investigate alternative designs involving multiple or hierarchical controllers for scalability. We are developing NS3 [14] simulations to study the scalability of XCo for thousands of nodes under multi-tiered multi-rooted data center topologies. Such simulations rely on accurate models of real-world network switches under congestion scenarios, which the current NS3 switch models do not provide.

Active and On-demand Coordination: While our current prototype coordinates the entire network, the XCo architecture is amenable to controlling congestion in an "active and on-demand" fashion. In other words, since central coordination is unnecessary when the network is uncongested, XCo can trigger central coordination only during times of network congestion. This requires accurate and proactive measurement and feedback of the traffic matrix to the central controller from potentially thousands of local coordinators. Even during congestion, the central controller need not impose coordination on all V2V flows. The local coordinators can measure and report only the large V2V flows, which can then be regulated by the central controller without limiting transmissions from other well-behaved flows. The rule of thumb is that central controller should attempt to impose minimal necessary constraints on transmissions and only to prevent congestion where necessary.

Fault Tolerance: Three major types of failures must be addressed in XCo: controller failure, end-host failure, and loss of transmission directives. Failure of a central controller(s) does not need to imply that network activity comes to a halt. If the local coordinators do not receive a transmission directive from a central controller for a certain time, they could switch to uncontrolled transmission to keep their network activities alive. Non-arrival of multiple successive directives could indicate to a local coordinator that either the central controller has failed, or that the network is partitioned. The central controller can also be mirrored by a hot-standby node to provide non-stop operations. Central controller can also delete V2V flows from its schedule if that node's local coordinator fails to send periodic feedback.

Alternative Coordination Strategies: The XCo framework admits of several alternative coordination strategies that could provide better performance. For instance, our current definition of the feasibility condition in timeslice scheduling implies that if any node x has an outgoing link capacity C_x that is greater than the bottleneck link capacity to a destination, then x cannot transmit during congestion. This limitation could be overcome by a hybrid of timeslice scheduling and rate-limiting in which each V2V flow can be rate-limited within individual timeslices. Other coordination strategies could enforce QoS among flows by allocating differentiated transmission rates and/or timeslice lengths.

7. CONCLUSION

This paper makes the case that virtualization offers new opportunities to alleviate congestion in data center Ethernet. We present the design, implementation, and evaluation of a prototype system, called XCo, that explicitly coordinates network transmissions from virtual machines across a cluster's Ethernet infrastructure. A central controller issues transmission directives to individual nodes at fine-grained timescales (every few milliseconds) to temporally separate transmissions competing for bottleneck links. We offer evidence through extensive evaluations that such explicit coordination has the potential to prevent congestion-induced performance problems in today's unmodified Gigabit and 10GigE switched Ethernet. Our techniques require no changes to the VMs, applications, protocols, or the network switches. Future work includes evaluating more complex topologies, scalability, on-demand coordination, fault-tolerance, and alternative coordination strategies.

Acknowledgements

We are highly grateful to our shepherd, Kenneth Yocum, for his invaluable input towards improving this paper. We also thank all anonymous reviewers for their suggestions. This work is supported by the National Science Foundation through awards CNS-0845832, CNS-0855204, and CNS-0751121.

8. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. of SIGCOMM 2008*, Aug. 2008.

[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proc. of Networked Systems Design and Implementation (NSDI) Symposium, San Jose, CA*, April 2010.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Symposium on Operating Systems Principles*, 2003.

[4] P. J. Braam. *File systems for clusters from a protocol perspective. http://www.lustre.org*.

[5] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *Proc. of NSDI*, 2005.

[6] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. *SIGCOMM Comput. Commun. Rev.*, 37(4):1–12, 2007.

[7] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP Incast throughput collapse in datacenter networks. In *Workshop on research on enterprise networking*, pages 73–82, 2009.

[8] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. of NSDI*, 2005.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of ACM*, 51(1):107–113, 2008.

[10] C. Diot and J.-Y. L. Boudec. Control of best effort traffic. *IEEE Network*, pages 14–15, May/June 2001.

[11] O. Feuser and A. Wenzel. On the effects of the ieee 802.3x flow control in full-duplex ethernet lans. In *Proc. of Local Computer Networks, Lowell, MA*, 1999.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5), 2003.

[13] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM*, 2009.

[14] T. R. Henderson, S. Roy, S. Floyd, and G. F. Riley. The NS-3 project. In *http://www.nsnam.org/*.

[15] IEEE 802.1. *802.1Q - Virtual LANs, http://www.ieee802.org/1/pages/802.1Q.html*.

[16] IEEE 802.1 Data Center Bridging Task Group. *http://www.ieee802.org/1/pages/dcbridges.html*.

[17] INCITS Technical Committee T11. *Fibre Channel over Ethernet, http://www.t11.org/fcoe*.

[18] Internet Small Computer Systems Interface (iSCSI). *http://tools.ietf.org/rfc/rfc3720.txt*.

[19] K. Kant. Towards a virtualized data center transport protocol. In *Workshop on High Speed Networks*, 2008.

[20] C. Kim, M. Caesar, and J. Rexford. Floodless in seattle: a scalable ethernet architecture for large enterprises. In *Proc. of the ACM SIGCOMM*, 2008.

[21] A. Kuzmanovic and E. W. Knightly. Low-rate TCP-targeted denial of service attacks: The shrew vs. the mice and elephants. In *SIGCOMM*, 2003.

[22] Linux Advanced Routing and Traffic Control. *http://lartc.org/howto/*.

[23] Memcached. *A distributed memory object caching system, http://memcached.org/*.

[24] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. Spain: Cots data-center ethernet for multipathing over arbitrary topologies. In *Proc. of Networked Systems Design and Implementation (NSDI) Symposium, San Jose, CA*, April 2010.

[25] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*, 2009.

[26] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale storage cluster: Delivering scalable high bandwidth storage. In *Proc. of Supercomputing*, 2004.

[27] Netperf. *http://www.netperf.org/netperf/*.

[28] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of tcp throughput collapse in cluster-based storage systems. In *Proc. of File and Storage Technologies*, pages 1–14, 2008.

[29] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren. Cloud control with distributed rate limiting. In *SIGCOMM*, 2007.

[30] V. S. Rajanna, S. Shah, A. Jahagirdar, and K. Gopalan. Xco: Explicit coordination for preventing congestion in data center ethernet. In *Proc. of International Workshop on Storage Network Architecture and Parallel I/Os*, May 2010.

[31] Scaling memcached at Facebook. *http://www.facebook.com/note.php?note_id=39391378919*.

[32] S. Sharma, K. Gopalan, S. Nanda, and T. cker Chiueh. Viking: A multi-spanning-tree ethernet architecture for metropolitan area and cluster networks. In *Proc. of IEEE Infocom, Hong Kong, China*, March 2004.

[33] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (NFS) version 4 protocol. Request for Comments - RFC 3530, April 2003.

[34] R. Stanojevic and R. Shorten. Generalized distributed rate limiting. In *Proc. of International Workshop on Quality of Service (IWQoS), Charleston, SC*, 2009.

[35] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *SIGCOMM*, 2009.

[36] H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. In *Proc. of IEEE*, volume 83(10), pages 1374–1396,, 1995.