Title	PVTCP: Towards practical and effective congestion control in virtualized datacenters
Author(s)	Cheng, L; Wang, CL; Lau, FCM
Citation	The 21st IEEE International Conference on Network Protocols (ICNP 2013), Göttingen, Germany, 7-10 October 2013. In International Conference on Network Protocols Proceedings, 2013, p. 1-10
Issued Date	2013
URL	http://hdl.handle.net/10722/191544
Rights	International Conference on Network Protocols. Proceedings. Copyright © Institute of Electrical and Electronics Engineers.

PVTCP: Towards Practical and Effective Congestion Control in Virtualized Datacenters

Luwei Cheng, Cho-Li Wang, Francis C. M. Lau
Department of Computer Science, The University of Hong Kong
{lwcheng, clwang, fcmlau}@cs.hku.hk

***Abstract*—While modern datacenters are increasingly adopting virtual machines (VMs) to provide elastic cloud services, they still rely on traditional TCP for congestion control. In virtualized datacenters, TCP endpoints are separated by a virtualization layer and subject to the intervention of the hypervisor's scheduling. Most previous attempts focused on tuning the *hypervisor layer* to try to improve the VMs' I/O performance, and there is very little work on how a VM's *guest OS* may help the *transport layer* to adapt to the virtualized environment. In this paper, we find that VM scheduling delays can heavily contaminate RTTs as sensed by VM senders, preventing TCP from correctly learning the physical network condition. After giving an account of the source of the problem, we propose PVTCP, a ParaVirtualized TCP to counter the distorted congestion information caused by VM scheduling on the sender side. PVTCP is self-contained, requiring no modification to the hypervisor. Experiments show that PVTCP is much more effective in addressing incast congestion in virtualized datacenters than standard TCP.**

I. Introduction

Cloud datacenters allow users to rent a cluster of VMs in an on-demand fashion and access them remotely. In datacenters, TCP is primarily and unequivocally used for communication. A recent study [4] reveals that 99.91% of traffic in today's datacenters is TCP traffic. Since its inception, TCP has been successfully deployed in: (i) various WAN environments, where the network delay can be tens or hundreds of milliseconds, and (ii) *physical* datacenters, featuring sub-millisecond network delay. However, as illustrated in Figure 1, whether TCP is able to function well in *virtualized* datacenters is largely an open question. Unlike in WANs and physical datacenters where the network delay is relatively stable and predicable, it has been observed that the network delay in virtualized datacenters can vary significantly and tends to be highly unpredictable [6], [32]. This unlikeness calls for a re-examination of TCP's effectiveness in virtualized execution environments.

TCP incast [27] is a particular form of network congestion which has become a bothersome issue in datacenter networks. It is commonly seen in large-scale distributed data processing such as those using MapReduce [12] and web search. Among virtualized clouds, Amazon for one is using VMs to provide Elastic MapReduce (EMR) service [1]. When multiple senders communicate with a single receiver via a bottleneck link, the highly synchronized traffic can quickly fill the limited output buffer of that switch port. Due to the tail-drop policy

978-1-4799-1270-4/13/$31.00

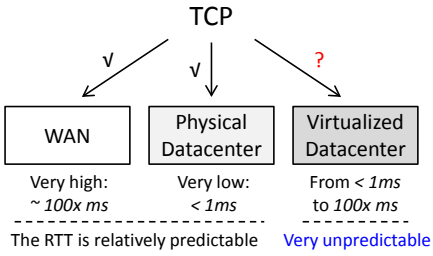


Fig. 1. Can TCP still work as well as before?

implemented in most commodity switches, late-coming packets will be abandoned, resulting in intensive packet loss as suffered by the TCP senders. And hence the receiver-perceived throughput (goodput) can be orders of magnitude lower than the link capacity. Despite much prior work on TCP in *physical* datacenters [16], [8], [7], a good understanding of how TCP behaves in *virtualized* datacenters is still lacking.

Previous studies on VMs' I/O performance mostly focused on modifying the hypervisor layer, whereas the study of how VM's transport-layer protocol can automatically adapt to this virtualized platform appears to have been overlooked. This paper provides three major contributions. First, a finding: We find that VM scheduling delays make TCP unable to correctly learn the network condition from round-trip time (RTT) measurements, causing serious disorder to standard congestion control methods. Second, we provide a concrete analysis to explain TCP's peculiar behaviors, and address the necessity of paravirtualizing TCP. Third, we propose PVTCP for the sender side, a practical solution to effectively filter out negative effect arising from VM scheduling delays. We have implemented our solution in Linux. The evaluation results show that PVTCP can: (1) avoid performance drop caused by pseudo-congestion from the hypervisor scheduler, and (2) avoid incast throughput collapse caused by multiple synchronized traffic patterns.

The rest of the paper is organized as follows. We introduce the background and bring out our motivation in §II. The experimental methodology is presented in §III. We describe TCP's dilemma in handling incast congestion in a virtual cluster in §IV. We then provide a concrete understanding of the problem in §V. Possible approaches are discussed in §VI. We propose our solution in §VII. Implementation and evaluation are presented in §VIII and §IX. The future work is discussed in §X and the related work in §XI.

II. Motivation and Background

A. Congestion Control – the Core of TCP

TCP is designed to be end-to-end: there is no global coordinator and each host relies on implicit signals to infer the condition of the network and then adjusts its own sending rate. Reliable transmission is achieved via the use of a retransmit timer: for the segments sent each time, the sender expects an ACK from the receiver before the timer expires; without receiving the ACK in time, some segment is considered to be lost, presumably due to network congestion and will be retransmitted at some appropriate instant later. A key to TCP's retransmission is the accurate RTT measurements. The sender measures the time between when data is sent and when the corresponding ACK returns. Standard TCP uses a low-pass filter to estimate the timeout values:

$$SRTT_i = \frac{7}{8}SRTT_{i-1} + \frac{1}{8}MRTT_i \quad (1)$$

$$RTTVAR_i = \frac{3}{4}RTTVAR_{i-1} + \frac{1}{4}|SRTT_i - MRTT_i| \quad (2)$$

$$RTO_{i+1} = SRTT_i + 4 \times RTTVAR_i \quad (3)$$

where $MRTT_i$ is the measured RTT at time i, calculated using successfully returned ACK packets, $SRTT_i$ is the smoothed RTT, and $RTTVAR_i$ represents the variance (mean deviation) of recent RTTs.

If ACK packets are returned quickly, the sending window will be increased by one segment each time in slow-start phase, allowing TCP to transmit data more quickly. If however an ACK packet does not return before the retransmit timer expires, the sender will double the RTO value for each consecutive timeout (exponential backoff):

$$RTO_{i+1} = RTO_i \times 2 \quad (4)$$

To protect the sender from spurious timeouts, standard TCP explicitly sets a lower bound, RTO_{min}. Linux adopts 200ms as the default value for RTO_{min}, which may be suitable for WANs. But for datacenters where delays are sub-millisecond, a small value (e.g. 1ms) has been proven to be more suitable when dealing with network congestion [31], [8].

B. Virtualization Makes a Difference

Virtualization technology is a key enabler of cloud computing. The hypervisor allows multiple VMs to run on a single physical host by multiplexing the underlying physical resources, such as CPU, memory and I/O. Figure 2 shows the conceptual organization of Xen [5], the most widely deployed open-source hypervisor. Xen provides basic mechanisms for the upper-layer domains, such as VM scheduling, event delivering, shared memory, etc. For safety reasons, guest domains are not allowed to access the hardware directly but rely on the driver domain (also called *domain zero*) to act on their behalf, which embraces real hardware drivers; this is known as the *split-driver* model. Take network traffic for example, the packets from the TCP layer are firstly delivered to the frontend (`netfront`, a layer-2 device), and then passed to

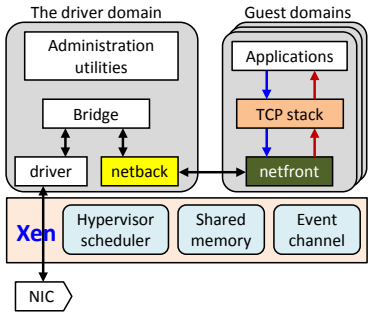


Fig. 2. Xen and its split-driver model for network.

the backend (`netback`, also a layer-2 device) residing in the driver domain; through the network bridge, the packets arrive at the real device driver and are eventually pushed to the physical network; and vice versa. A very important role of the hypervisor is to proportionally allocate CPU cycles among the VMs. The hypervisor scheduler guarantees that each VM receives a CPU allocation commensurate with its relative 'weight' as set by the administrator. Since the driver domain is responsible to forward I/O for all guest domains, in order to guarantee the efficiency, it often runs on dedicated CPU cores. In our experiments, we also follow this practice.

Virtualization can cause performance problems to applications, which do not exist when they directly run in physical machines. First, since all I/O traffic must go through the driver domain, extra software overhead is incurred. Second, hypervisor scheduling activities can significantly degrade a VM's I/O performance, because the VM's scheduling delays eventually translate into the processing delays of its I/O events.

III. Experimental Methodology

We did not use public cloud platforms like Amazon EC2, because they would not allow access to the hypervisor. We conduct the experiments in our own cluster emulating a cloud, in which we can completely control the VM settings and the guest OS kernels, so that the experimental phenomena can be reproduced in a deterministic way. The cluster contains 21 Dell PowerEdge M1000e blade servers, connected through a Brocade FastIron SuperX GbE switch. Each server is equipped with two quad-core 2.53GHz Intel Xeon 5540 CPUs, 16GB physical memory, and two 250GB SATA hard disks.

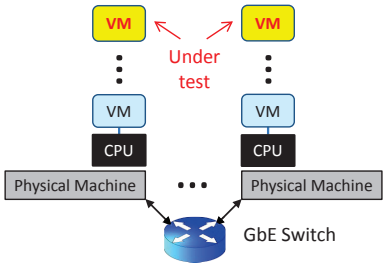
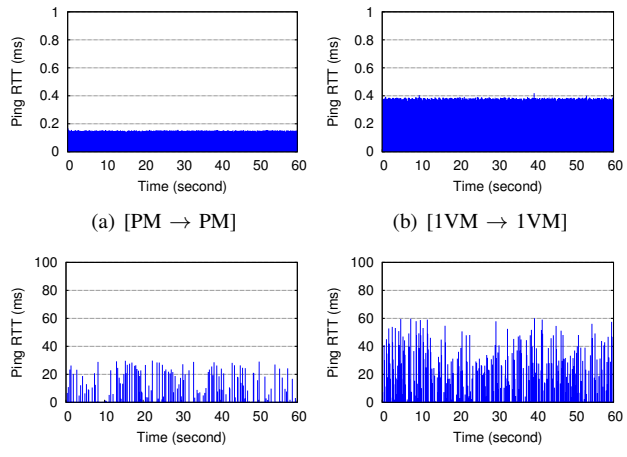


Fig. 3. Experimental setups

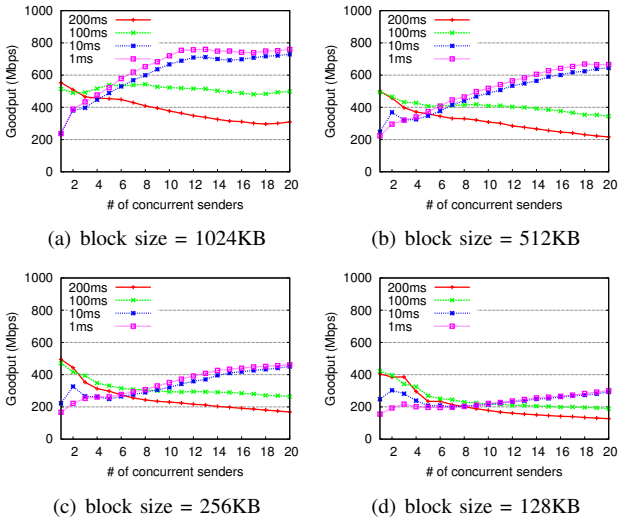
We avoid simulation because it is difficult to simulate so many non-deterministic factors, especially those due to the



(a) [PM → PM] (b) [1VM → 1VM]

(c) [1VM → 2 co-located VMs] (d) [1VM → 3 co-located VMs]

Fig. 4. How virtualization affects RTTs, with and without VM consolidation.



(a) block size = 1024KB (b) block size = 512KB

(c) block size = 256KB (d) block size = 128KB

Fig. 5. TCP incast in a virtual cluster, with different requested block sizes and different RTO_{min} values.

hypervisor. Our benchmark is identical to that in [31] as we have simply reused their source code[1]: the client issues a certain number of requests to several servers for data blocks; only after all responses of the current request are received will the client issue the next request. Without loss of generality, we load each CPU core with 2 or 3 co-located VMs to share the CPU cycles (Figure 3), which is a common loading in public cloud platforms. We use the Xen 4.1.2 hypervisor and Linux 2.6.32 with TCP NewReno implementation in the guest OS. We use `tcpdump` and `netstat` to collect corresponding information. Besides, `tcp_probe` kernel module is modified to observe the in-kernel TCP variables.

IV. Problem Description

A. RTTs in a Virtualized Datacenter

In this section, we separately examine the two types of delays as mentioned in §II-B: the delays caused by the running of the hypervisor software per se (when there is no CPU sharing), and the delays due to VM scheduling (when multiple VMs are consolidated in one CPU core). We use `ping` with a 0.1 second interval to get a fine-grained view, with each test lasting for 60 seconds. For consolidated VMs, each VM runs certain CPU workload to trigger the hypervisor scheduling.

Figure 4(a) shows that the average delay of the physical network (two hops, connected to the same switch) is 0.147ms. When both communicating hosts are VMs, as in Figure 4(b), the average RTT increases to 0.374ms (2.54×). The extra 0.227ms is the software overhead introduced by additional data movements between the driver domain and guest VMs. When one communicating VM runs with consolidation in Figure 4(c)(d), RTT largely varies without any apparent predictability. For example, with 3 co-located VMs per core in Figure 4(d), the maximum RTT is about 60ms.

Since the scheduling latency of each VM, Lat_{sched_vm}, is actually its queuing delay in the VM scheduling queue, the maximum value of Lat_{sched_vm} is:

$$max(Lat_{sched\_vm}) = (N-1) \times TS_{hypervisor} \quad (5)$$

where N is the number of co-located VMs on that core, and $TS_{hypervisor}$ is the time slice used in the hypervisor scheduler. Xen's scheduler [2] uses a 30ms time slice by default. So when N is 3, the maximum scheduling delay of each VM is 60ms.

From the observations above, we have two findings: (1) virtualization does incur some software overhead, increasing RTT by several hundred microseconds; (2) even when there is no congestion, the VM scheduling can make RTT fluctuate heavily, with the peak linearly increasing with the number of co-located VMs. Comparing the two types of delays, VM scheduling latency is the dominant factor affecting RTT.

B. TCP Incast in a Virtual Cluster

For a physical cluster, previous studies [27], [8] have examined several techniques (limited transmit, reduced duplicate ACK threshold, disabling slow-start, randomizing timeout values) and several lost-recovery variants (Reno, NewReno, SACK), but they concluded that none of them can fully eliminate the incast throughput collapse. On the other hand, significantly reducing RTO_{min} has been shown to be a safe and effective approach [31]. However, whether this approach is still effective in a virtual cluster is still unknown. In this section, we add competing flows using multiple VMs to investigate how TCP would react when real network congestion happens together with spiked RTTs. There are 3 co-located VMs per core (including the sender VM), with background VMs running certain CPU workload to trigger hypervisor scheduling. The receiver VM runs on a dedicated core (more discussions are in §X). In each group test, the receiver requests the same amount of data from the senders: 2000 blocks for 1024KB size, 4000 blocks for 512KB size, 8000 blocks for 256KB size and 16000 blocks for 128KB size.

[1]Available at: https://github.com/amarp/incast.

TABLE I
TO STROBE 2000 1024KB BLOCKS FROM ONE VM SENDER.

RTO_{min}	200ms	100ms	10ms	1ms
Goodput (Mbps)	559.7	517.2	256.4	234.2
Time taken (seconds)	32	34	66	72
Segments sent	42756	53139	104242	106140
Retransmitted	0	39	587	698
TCP timeouts	0	17	587	698
Spurious RTOs	0	12	108	105
Duplicate ACKs	0	14	584	694

Figure 5 shows that although TCP-1ms causes very serious performance degradation at the beginning, when the number of concurrent senders increases, surprisingly the performance gap shrinks and eventually TCP-1ms remarkably outperforms TCP-200ms. For example, in the cases of 20 senders, TCP-1ms achieves more goodput: 102.8% for 1024KB, 188.4% for 512KB, 225.8% for 256KB and 198.7% for 128KB respectively. This *catch-up-then-surpass* phenomenon has also been observed in other experiments with different VM densities.

V. UNDERSTANDING THE PROBLEM

A. One-to-one Communication

Among the experiments above, we first select the cases of one sender, as in Figure 5(a), for deeper investigation. The results are shown in Table I. First, a large number of TCP timeouts happen when using small RTO_{min} values, resulting in a great deal of retransmissions. The smaller the RTO_{min} is, the more timeouts the sender experiences. Second, with TCP-1ms and TCP-10ms, only a small part of timeouts are identified to be spurious by F-RTO [29]. Intuitively, *all* timeouts should be spurious because no packet actually gets lost when there is only one sender. Third, the number of duplicate ACKs received is almost the same with the number of timeouts.

Compared with the overall throughput, the extra network load due to retransmissions is small. What is more serious is the reduction of the sending window after RTOs. Upon a timeout event, the sending window will be reduced to 1 MSS (maximum segment size). If the RTO is detected to be spurious, Linux TCP typically has two policies to respond: (1) a conservative (default) response is *rate halving*, namely to reduce the congestion window (`cwnd`) and the slow-start threshold (`ssthresh`) by half; (2) a more aggressive response is to *undo* the congestion control by restoring `cwnd` and `ssthresh` to the values before timeout. In Figure 6, we examine the effectiveness of the two policies. The *undo* policy can indeed improve the VM's sending performance to some extent, but when comparing with TCP-200ms, the performance gap is still not negligible.

1) ***What happens in the guest OS****:* Figure 7 presents a schematic explanation for RTOs in the sender VM. The vertical dimension is time and the horizontal dimension is flow direction. Supposing that VM1 is the TCP sender, after sending a number of data packets, if the receiver can return the ACK packet within VM1's scheduling time slice: (1) the retransmit timer will be cleared and reset with a new timeout value; (2) the sending window will be doubled if TCP is in the slow-start

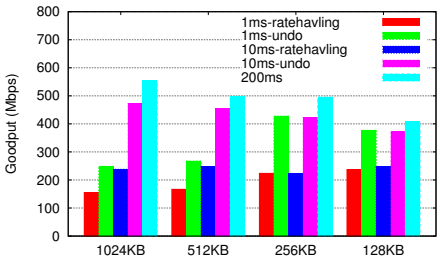


Fig. 6. The effectiveness of two response policies to spurious RTOs.

phase. Otherwise, if VM1 has been preempted, the ACK will be buffered in the driver domain (VM1's netback). Later when VM1 receives CPU cycles again, it will check all pending interrupts: (1) the guest OS updates the system clock and runs the expired timers – immediately RTO happens; (2) soon after the RTO, the ACK is received by VM1's netfront via network interrupt. Due to common OS design, since kernel timers are used to create timed events for many other kernel services, such as process scheduler and networking, timer interrupt is always executed *before* other interrupts. In contrast, much of the network's processing is executed a bit later using bottom-half. Once RTO happens, TCP assumes that there is serious network congestion, but actually the congestion sensed by the VM is *pseudo-congestion*[2].

To verify our analysis in Figure 7, we show a micro-view of TCP's behavior in Figure 8 by capturing the picture around the happening of VM scheduling delays. In steps 2 and 3, even though the receiver has returned the ACKs (stored in the driver domain of the sender machine), after the sender VM wakes up, RTO is still triggered. If there are no less than *two* consecutive ACKs that can all advance `snd_una`, F-RTO uses this heuristic to declare the timeout to be spurious [29]. But the number of ACKs that will be returned each time is uncertain, subject to the receiver's situation, e.g. available buffer space, delayed ACK, etc. Therefore F-RTO is not able to identify all spurious timeouts. As for the receiver, once receiving the old segment sent in step 3, it returns an duplicate ACK to the sender in step 4. Both RTO and duplicate ACKs are detrimental to TCP's sending speed.

Figure 7 and Figure 8 only reveals the problem on the sender side. In fact, hypervisor scheduling can also delay the return of ACK packets on the receiver side, but the situation there is different, which we will discuss in §X. In this paper, we mainly focus on understanding how the sender side is affected by VM scheduling delays.

2) ***Why*** $RTTVAR$ ***can not adapt****:* A spurious timeout occurs when RTT suddenly increases and exceeds the retransmit timeout value that has been determined previously. The low-pass filter used by standard TCP was developed according to queuing theory that predicts the delays from the *physical network*. However, hypervisor scheduling is totally transparent to the VMs. Once it suddenly happens, the added delay can

[2]In the rest of this paper, we use this term to refer to the RTOs caused by the VM scheduling delays from the hypervisor.

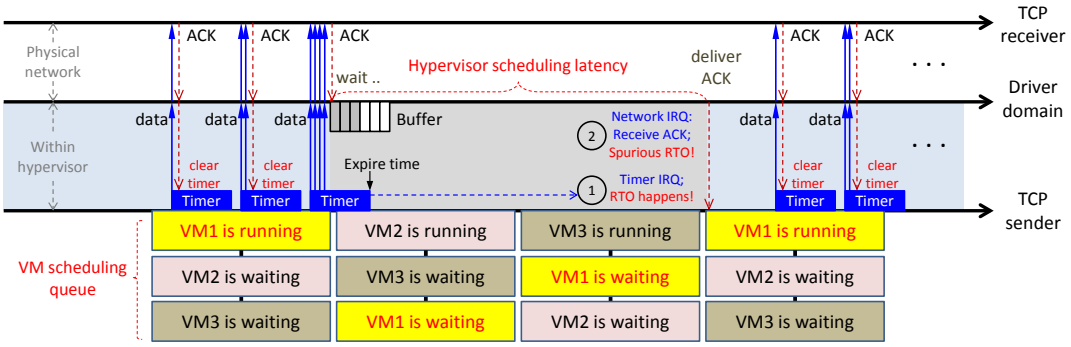


Fig. 7. How pseudo-congestion happens in the guest operating system when the TCP sender is a consolidated VM.

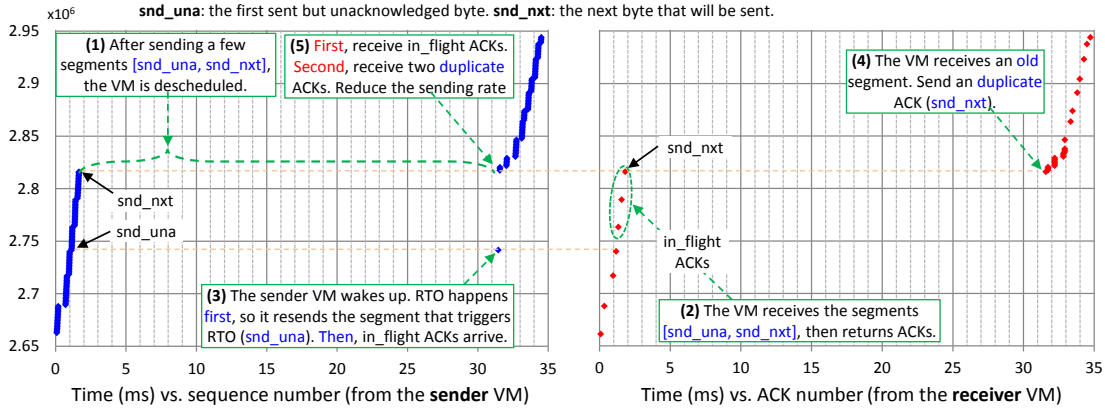
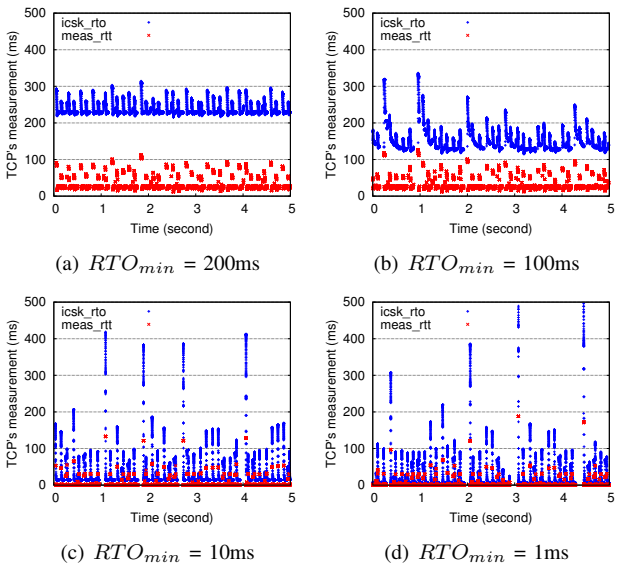


Fig. 8. How TCP behaves with a small RTO_{min}, when there is CPU sharing on the sender side.



(a) RTO_{min} = 200ms (b) RTO_{min} = 100ms

(c) RTO_{min} = 10ms (d) RTO_{min} = 1ms

Fig. 9. The measured RTTs and calculated RTO values in a sender VM when varying RTO_{min}.

be hundreds of times of the physical network delay. RTT estimation algorithm thus has no chance to react to it timely.

In Figure 9, we use our modified `tcp_probe` kernel module to report the measured RTTs (`meas_rtt`) and the calculated RTO values (`icsk_rto`) in the sender VM when altering RTO_{min}. We can see that RTT spikes always appear

TABLE II
TO STROBE 40GB DATA FROM 20 VM SENDERS (1024KB BLOCK SIZE).

RTO_{min}	200ms	100ms	10ms	1ms
Goodput (Mbps)	309.5	498.6	729.3	759.9
Time taken (seconds)	1058	656	450	424
Retransmitted	137478	141642	68448	67604
TCP timeouts	26060	27181	50967	59182
Spurious RTOs	0	377	27032	31415
Duplicate ACKs	26134	27957	49452	58392

without any prior prediction from its historical measurements. The variable $RTTVAR_i$ in Equation 2 can only reflect the variance of *previously measured* RTTs. Without the protection of a sufficiently large RTO_{min}, TCP timeout happens *before* the current $RTTVAR$ can adapt to the change. Upon each RTO, TCP would exponentially increase subsequent timeout values using Equation 4, e.g. in Figure 9 (c)(d).

B. Many-to-one Communication

We further select the cases of 20 concurrent VM senders, as in Figure 5(a), for deeper investigation. Table II shows the collective statistical results from 20 VM senders. With TCP-200ms, TCP timeout happens much less frequently than TCP-1ms (56% less). However, the number of retransmitted segments is 103.4% more than that of TCP-1ms. Even though 53.1% of the timeouts in the TCP-1ms case are detected to be spurious, the goodput in this case is much higher than that of TCP-200ms (145.5% more). This proves that even in a virtual

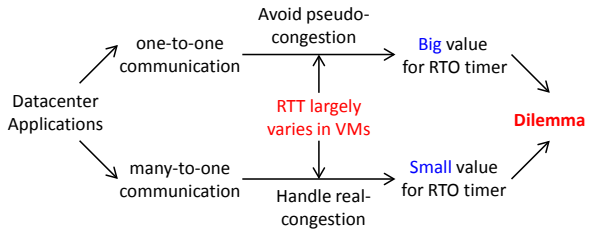


Fig. 10. TCP's inability to effectively function in virtualized datacenters

cluster, a small RTO_{min} is still necessary. Although it would make VMs vulnerable to pseudo-congestion, when there is heavy network congestion with many concurrent senders, the determining factor is whether the VM can sense the packet loss as quickly as possible and perform retransmissions in time.

C. Summary

Hypervisor scheduling can cause RTT spikes, even when there is no congestion in the physical network. Since the delay variability induces serious *pseudo-congestion*, one effective way to solve the problem is to use a sufficiently large RTO_{min} (say 200ms). However, when *real congestion* happens, a large RTO_{min} makes TCP senders numb to packet losses, leading to stalled flows for a long time and thus low goodput. The more aggressive RTO_{min} is, the less time a connection spends waiting for needed RTOs, but meanwhile it experiences more spurious ones. It seems impossible to determine a *good* RTO_{min} which can work well for all scenarios. The two contradictory requirements present a dilemma to VM's TCP design, as illustrated in Figure 10.

This problem is not limited to a specific congestion control algorithm, such as Reno, BIC, CUBIC, Vegas, etc. Virtualization shakes the foundation upon which all TCP variants have been built: RTT can no longer be a reliable indicator to infer the network condition, because the measured RTTs may have been contaminated by VM scheduling delays.

VI. Possible Approaches

A. Hypervisor-level Solutions?

One possible solution to improve a VM's TCP performance which is affected by VM scheduling is to modify the hypervisor scheduler: when network packets arrive (either data packets arriving on the receiver's side or ACKs arriving on the sender's side), schedule the concerned VM as soon as possible as in [26], [19], [10], [34], regardless of its priority, state, credits, etc. This approach can alleviate the problem to some extent, since obviously it would shorten the scheduling delays of the VM. It would however introduce a different problem: the VM context switching overhead in the hypervisor will increase substantially, because the hypervisor needs to keep swapping the VMs in response to incoming I/O events, making these solutions too expensive in practice. We are aware that the vCPU switching of SMP VMs can be less frequent by properly migrating interrupts from a preempted vCPU to a running one inside the guest OS [9], but VMs with only a single vCPU will not be able to enjoy this benefit.

In virtualized clouds, in fact, VM scheduling needs to happen in order to properly and fairly share the CPU, and should not be perturbed by ordinary I/O events. As the hypervisor scheduler is used very frequently to manage all the VMs, it is important that its design and implementation be as simplistic and focused as possible: the main function of the hypervisor scheduler is to proportionally allocate CPU time among the VMs, and not so much to satisfy VM's I/O requirements. Therefore, our direction is to stay away from the hypervisor and try to redesign the upper-layer protocols to automatically tolerate the VM scheduling delays.

B. Other TCP Variants?

RTT spikes can also appear in other network types and cause spurious timeouts, for example and notably in wireless environments. In these networks, high bit error rate is another main cause of packet loss, aside from network congestion. Since there are many different access technologies for wireless networks, each TCP wireless solution tends to have its own unique problems to tackle. For example, TCP-Peach [3] considers long propagation delay in satellite networks, ATCP [20] solves the problem of frequent route changes in ad-hoc networks, and Freeze-TCP [14] focuses on the hand-off problem in cellular networks. It is unlikely that a universal TCP solution can be developed that fits all types of networks [30]. Seemingly, to accurately identify *the cause of packet loss* has become a focal point in TCP design. Virtualized datacenters, because of their unique causes of unpredictable network delays of large magnitude (100×) and high frequency (every 10×ms), present another case needing special adaptation.

C. Timestamping the Packets in the Device Driver?

As shown in Figures 7 and 8, the sender VM can not know the arrival of the ACK until the sender is scheduled again. The time that the sender receives the ACK is not the time that the ACK arrives at the sender's machine. To solve the problem, one would be tempted to try correcting the timestamp of the ACK on reception in the device driver and exposing the value to the guest OS. TCP can use a timestamp option to measure RTT: the sender timestamps the transmission, and the receiver will *echo* back this value via its ACK to the sender. Upon receiving the ACK, the sender calculates the difference between the current system clock and the ACK's timestamp:

$$MRTT = system\_clock - ACK.timestamp \quad (6)$$

So $ACK.timestamp$ is actually the *sending time* of last transmission from the sender, and not the *arriving time* of the ACK. As for $system\_clock$, it can be obtained only inside the guest OS after the VM wakes up. Therefore, changing the ACK timestamp in the device driver will easily end up faulty.

VII. Proposed Solution – PVTCP

PVTCP is a **P**ara**V**irtualized TCP that does not require any modification to the hypervisor. It *accepts* the latencies thus introduced by the hypervisor scheduler as they are, but suggests a way to capture the *true* picture of every transmission

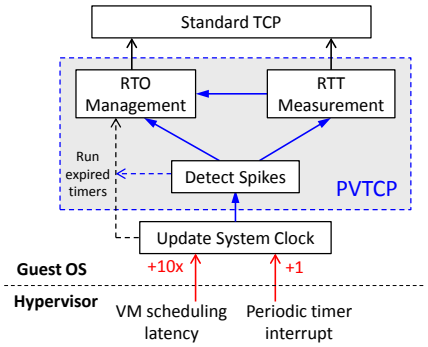


Fig. 11. The architecture of PVTCP.

involving these latencies, which is then used to determine more accurate RTO values that can help filter out pseudo-congestion.

A. The Architecture of PVTCP

Figure 11 shows the architecture of PVTCP. Since pseudo-congestion only happens when the sender VM just wakes up, if we can detect its wakeup, we have the chance to deal with the problem. The retransmit timer relies on the system clock (`jiffies` in Linux) for its expiry time. RTT is also calculated using the system clock. In a consolidated VM, there are two sources in the hypervisor that contribute to the VM's clock ticks: VM scheduling latency and periodic timer interrupt. The latter is very regular and increases `jiffies` by one each time, whereas VM scheduling latency is very unpredictable and can cause sudden increase to `jiffies`. Hypervisor scheduling disrupts the guest OS's updating of its system clock. This problem is generic and not specific to Xen/Linux. Therefore, our key idea is to detect these sudden changes (spikes) of `jiffies`, and use this in a heuristic to filter out the negative effect of the spikes.

*1) **Detect Spiked Jiffies**:* The kernel increases the system clock by one at each timer interrupt. There are `HZ` timer interrupts in a second. In modern operating systems, especially 64-bit ones, `HZ` is mostly 1000 for precise time accounting, and `jiffies` is incremented every 1ms. Xen hypervisor uses a "one-shot" timer to provide the clock source to VMs. Normally when a VM is running, it can periodically register clock events and receive virtual timer IRQs from the hypervisor to continuously update its `jiffies`. However, if the VM has been preempted, it is unable to set clock events to the hypervisor and the pre-registered clock event has to be backlogged until the VM resumes getting CPU cycles, as shown in Figure 12. When there are three VMs sharing one CPU core, the maximum scheduling delay can be 60ms, resulting in a sudden change of 60 increments to `jiffies`. Therefore, if `jiffies` is detected to be *not continuously* increasing, it is a strong indication that there was a backlogged timer interrupt and the VM has *just* been delayed by the hypervisor scheduler.

*2) **RTO Management**:* Spurious RTOs may occur when the sender VM has just experienced a delay from the hypervisor scheduler. Suppose that TCP's retransmit timer is activated at

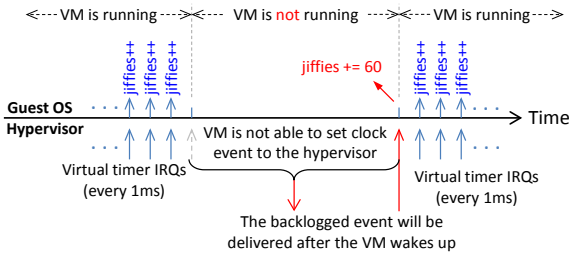


Fig. 12. How `jiffies` changes in a VM, when there is CPU sharing.

time T_{begin}, and it is set to expire at time T_{end}; then its active period can be expressed as $[T_{begin}, T_{end}]$. In the case of no scheduling delays from the hypervisor:

$$T_{end} = T_{begin} + RTO \tag{7}$$

Otherwise, the timer's expiry time would be postponed:

$$T_{end} = T_{begin} + max(Lat_{sched\_vm}, RTO) \tag{8}$$

In order to avoid the problem of "RTO-ACK-SpuriosRTO", PVTCP slightly extends the expiry time of the retransmit timer by 1ms, if a spiked `jiffies` has been detected:

$$T_{end} \leftarrow T_{begin} + max(Lat_{sched\_vm}, RTO) + 1ms \tag{9}$$

In this way, if an ACK does arrive in the driver domain, it gets an opportunity to reach the VM sender *before* RTO happens. After the ACK is received, the retransmit timer will be reset with a new timeout value. If there is no ACK packet in the driver domain due to real network congestion, TCP can know the packet loss at `jiffies+1` via RTO event, and then will retransmit the lost segments. Compared with the delays from the hypervisor scheduler (10×ms), the overhead of this temporary 1ms postponement is negligible.

*3) **RTT Measurement**:* With the RTO being avoided in the above, the received ACK will be used to calculate RTT in the sender VM. However, its arrival has been delayed by the hypervisor. Therefore, the measured RTT can not truly reflect the congestion condition of the physical network:

$$MRTT = max(Lat_{sched\_vm}, trueRTT) \tag{10}$$

Since Lat_{sched_vm} is usually 10×ms, it can cause acute increase to both $SRTT$ and $RTTVAR$ (Equation 1 and 2). As a result, the calculated timeout value for RTO will seriously deviate from the reasonable value (Equation 3). In our solution, since we are able to detect the moment when a spiked RTT happens, the untrustful measurement can be identified and then filtered out. PVTCP adopts a conservative way to measure RTT when the VM experiences hypervisor scheduling delays, by reusing the previously calculated "smoothed RTT":

$$MRTT_i \leftarrow SRTT_{i-1} \tag{11}$$

VIII. Implementation

PVTCP has been implemented in Linux 2.6.32, containing less than 200 lines of code. The design is generic and hence potentially portable to other types of guest operating systems.

TABLE III
THE IMPLEMENTATION OF PVTCP IN LINUX.

PVTCP Components	Where in Linux
Detect Spiked Jiffies	`run_timer_softirq()`
RTO Management	`__run_timers()`
RTT Measurement	`tcp_rtt_estimator()`

Algorithm 1: PVTCP algorithm

Global variables: `jiffies`, `prev_jiffies`, `is_spiked_jiffies`;

DetectSpikedJiffies()
begin
  for *each virtual timer interrupt in the guest OS* **do**
    if $jiffies > prev\_jiffies+1$ **then**
      `is_spiked_jiffies` $\leftarrow true$;
    else
      `is_spiked_jiffies` $\leftarrow false$;
    end
    `prev_jiffies` $\leftarrow$ `jiffies`;
  end
end

RTOManagement()
begin
  for *each timer softirq in the guest OS* **do**
    if `is_spiked_jiffies` *is true* **then**
      Identify all of TCP's retransmit timers;
      Reset them to expire at `jiffies+1`;
    end
    Run the other expired timers;
  end
end

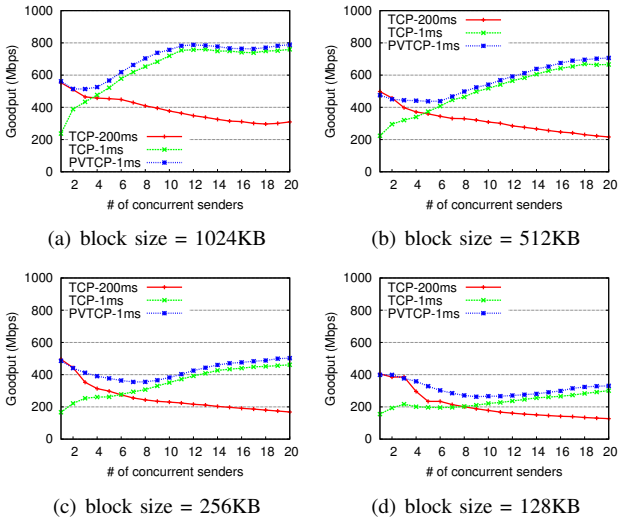
RTTMeasurement()
begin
  for *each received ACK packet* **do**
    if `is_spiked_jiffies` *is true* **then**
      $mearsured\_rtt_i \leftarrow smoothed\_rtt_{i-1}$;
    else
      Follow standard TCP to measure RTT;
    end
  end
end

The implementation aims to minimize the footprint by reusing existing Linux code as much as possible. The components of PVTCP are implemented as hook functions and are highly modular in Linux, as shown in Table III.

As illustrated in Algorithm 1, PVTCP is triggered only after a spiked jiffies has been detected in the guest OS; otherwise, it will adaptively switch to standard TCP for congestion control. The access to `is_spiked_jiffies` is protected by a spinlock to avoid race condition. We add a `type` variable in `struct timer_list`, and this variable will be set to `TCP_RETRANS` when TCP initializes its timers in `tcp_init_xmit_timers()`. In this way, we are able to distinguish exactly which timers should be slightly postponed when spiked jiffies happen, without affecting the other timers. It is possible that there are more than one retransmit timer belonging to multiple TCP connections; they are therefore put in a separate `struct list_head` for easy manipulation.

IX. PERFORMANCE EVALUATION

Recall in §IV that it is difficult to choose a suitable RTO_{min} that can fit the range of scenarios: too small a value (1ms) will suffer from pseudo-congestion in those one-to-one cases, but too large a value (200ms) will miss the real congestion in the many-to-one cases. We repeated the previous experiments with the addition of PVTCP, as shown in Figure 13, where we



(a) block size = 1024KB (b) block size = 512KB

(c) block size = 256KB (d) block size = 128KB

Fig. 13. PVTCP in a virtual cluster to handle incast congestion.

TABLE IV
PVTCP STATISTICS – TO STROBE 2GB DATA FROM ONE VM SENDER USING DIFFERENT BLOCK SIZES.

PVTCP-1ms	1024KB	512KB	256KB	128KB
Goodput (Mbps)	561.1	487.2	475.1	392.1
Time taken (seconds)	30.2	33.8	36.6	41.8
Retransmitted	7	10	13	11
TCP timeouts	2	3	3	6
Spurious RTOs	2	3	2	4
Duplicate ACKs	3	2	2	2

can see that PVTCP-1ms outperforms both TCP-200ms and TCP-1ms in almost all cases. More specifically, (1) PVTCP effectively avoids performance loss as happened in TCP-1ms with a small number of senders, and at the same time (2) it also avoids throughput collapse as experienced by TCP-200ms under heavy network congestion with a large number of senders. This benefit is important in practice as users can be freed from the worry of an improper RTO_{min}.

With one VM sender when there is no network congestion, the performance of PVTCP-1ms nearly coincides with that of TCP-200ms. The results in Table IV show that RTO happens rarely. Take blocksize of 1024KB for example, the number of timeouts is reduced from 698 (in Table I) to 2. A spurious RTO happens when the VM has just come out of a scheduling delay, and since PVTCP is able to detect this moment, it causes the ACK in the driver domain enter the VM slightly before the TCP timer expires.

Even when comparing with TCP-1ms in the case of 20 senders, the results in Table V show that PVTCP-1ms can still deliver a performance improvement of 3.6%, 5.8%, 7.9%, and 9.4% for the various blocksizes respectively. The numbers of RTOs, spurious RTOs, retransmissions and duplicate ACKs are all considerably reduced. This is because when real network congestion happens together with spiked RTTs, PVTCP is able to distinguish RTOs caused by pseudo-congestion from those caused by real packet losses. By weeding the bogus RTOs, the sender's congestion window suffers much less reduction.

TABLE V
PVTCP VS. TCP: TO STROBE 40GB DATA FROM 20 VM SENDERS CONCURRENTLY, USING DIFFERENT BLOCK SIZES

RTO_{min}=1ms	block size = 1024KB			block size = 512KB			block size = 256KB			block size = 128KB		
	TCP	PVTCP	**Gain**	TCP	PVTCP	**Gain**	TCP	PVTCP	**Gain**	TCP	PVTCP	**Gain**
Goodput (Mbps)	759.9	787.1	+3.6%	667.1	706.1	+5.8%	464.6	501.2	+7.9%	301.2	329.5	+9.4%
Time (seconds)	424	411	-3.1%	476	450	-5.5%	684	638	-6.7%	1067	998	-6.5%
Retransmitted	67604	65325	-3.4%	55969	46022	-17.8%	75801	17705	-76.6%	104778	36495	-65.2%
TCP timeouts	59182	26828	-54.7%	51408	15984	-68.9%	74672	17161	-77.0%	102461	36161	-64.7%
Spurious RTOs	31415	233	-99.3%	23303	253	-98.9%	22630	423	-98.1%	20120	346	-98.3%
Duplicate ACKs	58392	18822	-67.8%	47740	7729	-83.8%	61832	1596	-97.4%	72167	1620	-97.8%
DupACKs/Retrans.	86.4%	28.8%	-57.6%	85.3%	16.8%	-68.5%	81.6%	9.0%	-72.6%	68.9%	4.4%	-64.5%

Besides, since the old segments sent to the receiver caused by the bogus RTOs are also largely avoided, duplicate ACKs being returned to the senders are much reduced. The last row shows the ratios of duplicated ACKs to retransmissions, which clearly indicate that PVTCP is more capable of making *valid* retransmissions.

X. DISCUSSION AND FUTURE WORK

A. Sender-side vs. Receiver-side

Although VM scheduling can happen on both the sender's side and the receiver's side, the natures of the two situations are quite different. From TCP's perspective, and supposing a scheduling delay on either the sender side or the receiver side has caused an RTO: (1) if the sending VM has been preempted, RTO should *not* happen after the VM wakes up, because the ACK does arrive at the sender's side before the retransmit timer expires; (2) in contrast, if the receiving VM suffers the delay, RTO *should* happen in the sender because the generation and the return of the ACK is indeed too late.

The problem on the sender side is more an *OS problem* than a networking problem, because RTOs can be caused by the simple fact that timer interrupt is always executed *before* network interrupt. Our solution is not about how to *detect* the wrong RTOs or how to *recover* the congestion window after RTOs have happened. Instead, we aim at how to *avoid* the wrong RTOs, by giving the ACK a chance to be seen by TCP before the retransmit timer expires. This way, TCP does not need to go through a complicated path to handle RTOs at all.

We perceive the situation on the receiver side as a non-OS problem, by viewing the scheduling delay on the receiver side as just a part of the overall transmission "delay" which happens to have a large variance, and is similar to that of an unstable wireless path. But whether the guest OS on the receiver's side should or can do anything at all about this scheduling delay or not is an open problem, which is also the next step of our project.

B. The Principle of Packet Conservation

A very important principle of TCP design is "packet conservation": a new packet will not be pushed into the network until an old packet has left. The window-based transport protocol expects that each connection reaches an equilibrium using slow-start, and then counts on RTT (or ACK clock) to maintain the equilibrium [15]. However, the transmission of a VM is *intermittent* due to the scheduling delays, and RTT varies considerably with high frequency in virtualized datacenters. It has been proved that TCP has equilibrium and stability problems showing large and unstable network delays [21]. Therefore, all original models developed upon this principle need to be reexamined in virtualized datacenters, including RTT estimation algorithms, exponential backoff, the heuristics for fast retransmit and fast recovery, etc.

XI. RELATED WORK

A. TCP in Virtual Machines

vSnoop [17] lets the driver domain acknowledge the sender on behalf of the receiver VM, so as to elicit the sender to issue more packets. vFlood [13] encourages the sender VM to opportunistically flood as many packets as it can to the driver domain, as a way to make compensation for the period when the sender VM is descheduled and can not send packets.

First, the essence of both vSnoop and vFlood is to allow the guest VM to consume *more* resource from the driver domain, so that the TCP flow can still progress even when the VM has been descheduled. Instead, our solution avoids violating the resource allocation scheme in which each VM is supposed to consume a *fixed* amount of computing resources. Second, their solutions are typically of split-TCP [18] style, by dividing the TCP connection between VMs into two separate connections with the driver domain serving as the proxy point. A major problem is that the idea violates the end-to-end security protection between the sender and receiver. Third, their implementations are technically too complex because all TCP flows need to be monitored and manipulated in Xen's driver domain, making it less portable. In contrast, PVTCP is self-contained and highly hypervisor-independent.

Research [25] proposes to offload the whole socket layer (`open, accept, read, write, close`) into the hypervisor. However, this would greatly complicate a key feature of virtualization: VM live migration [11], because it would introduce a problem similar to that of residual dependencies in process migration [22]. PVTCP is self-contained, and therefore it will not be an obstacle to VM live migration.

B. Incast in Datacenters

This phenomenon was first described in the context of a distributed storage cluster [23]. Increasing the switch buffer size was one adopted solution, which however is too expensive. On the other hand, any particular switch configuration will have some maximum number of servers that can send simultaneously before throughput collapse occurs [27].

Application-level solutions were proposed in [28], [24], which requires building applications that conform to new programming interfaces. Although the synchronization pattern originates from the application layer, the problem actually happens in the transport layer. Since a datacenter needs to support a large number of applications, a solution at the transport layer is preferred.

ICTCP [33] adjusts the receive window to limit the bandwidth of each TCP sender before incast happens. However, under the anonymous sharing of the network, since each host has no idea of how the others behave, network congestion is essentially inevitable. Therefore, timeout-based solution on the sender side is still irreplaceable for dealing with packet loss.

DCTCP [4] leverages ECN capability in modern switches to mark congested packets. Both TCP sender and receiver are slightly modified for a fine-grained congestion window adjustment. Their experiments show that when the number of concurrent senders increases, DCTCP still needs to work with a small timeout value to handle heavy network congestion.

In virtualized datacenters, due to fluctuating delays caused by the hypervisor scheduler, TCP in VMs can not obtain trustable congestion information of the physical network from RTT feedbacks and RTO events. Therefore, none of the solutions designed for *physical* datacenters can be directly applied to *virtualized* datacenters without addressing the problem of VM scheduling delays.

XII. Acknowledgements

We thank the authors of [31] for sharing their source code for evaluating the incast problem. This research is supported in part by a Hong Kong UGC Special Equipment Grant (SEG-HKU09).

References

[1] Amazon Elastic MapReduce: http://aws.amazon.com/elasticmapreduce/.

[2] Xen's Credit Scheduler: http://wiki.xen.org/wiki/credit_scheduler.

[3] I. F. Akyildiz, G. Morabito, and S. Palazzo. TCP-Peach: a new congestion control scheme for satellite IP networks. *IEEE/ACM Transactions on Networking (ToN)*, 9(3):307–321, 2001.

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

[5] 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 *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[6] S. K. Barker and P. Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *ACM SIGMM Conference on Multimedia Systems (MMSys)*, 2010.

[7] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. *SIGCOMM Comput. Commun. Rev. (CCR)*, 40(1):92–99, Jan. 2010.

[8] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *ACM Workshop on Research on Enterprise Networking (WREN)*, 2009.

[9] L. Cheng and C.-L. Wang. vBalance: using interrupt load balance to improve I/O performance for SMP virtual machines. In *ACM Symposium on Cloud Computing (SoCC)*, 2012.

[10] L. Cheng, C.-L. Wang, and S. Di. Defeating network jitter for virtual machines. In *IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, 2011.

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

[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[13] S. Gamage, A. Kangarlou, R. R. Kompella, and D. Xu. Opportunistic flooding to improve TCP transmit performance in virtualized clouds. In *ACM Symposium on Cloud Computing (SoCC)*, 2011.

[14] T. Goff, J. Moronski, D. S. Phatak, and V. Gupta. Freeze-TCP: A true end-to-end TCP enhancement mechanism for mobile environments. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2000.

[15] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM Conference*, 1988.

[16] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *ACM Internet Measurement Conference (IMC)*, 2009.

[17] A. Kangarlou, S. Gamage, R. R. Kompella, and D. Xu. vSnoop: Improving TCP throughput in virtualized environments via acknowledgement offload. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[18] S. Kopparty, S. Krishnamurthy, M. Faloutsos, and S. Tripathi. Split TCP for mobile ad hoc networks. In *IEEE Global Telecommunications Conference (GLOBECOM)*, 2002.

[19] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik. Supporting soft real-time tasks in the xen hypervisor. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2010.

[20] J. Liu and S. Singh. ATCP: TCP for mobile ad hoc networks. *IEEE Journal on Selected Areas in Communications (JSAC)*, 19(7):1300–1315, 2001.

[21] S. H. Low, F. Paganini, J. Wang, S. Adlakha, and J. C. Doyle. Dynamics of TCP/RED and a scalable control. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2002.

[22] D. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, 2000.

[23] D. Nagle, D. Serenyi, and A. Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *ACM/IEEE Conference on Supercomputing (SC)*, 2004.

[24] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[25] A. Nordal, A. Kvalnes, and D. Johansen. Paravirtualizing TCP. In *International Workshop on Virtualization Technologies in Distributed Computing (VTDC)*, 2012.

[26] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling I/O in virtual machine monitors. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2008.

[27] A. Phanishayee, E. Krevat, V. Vasudevan, D. Andersen, G. Ganger, G. Gibson, and S. Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[28] M. Podlesny and C. Williamson. Solving the TCP-incast problem with application-level scheduling. In *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2012.

[29] P. Sarolahti, M. Kojo, and K. E. E. Raatikainen. F-RTO: an enhanced recovery algorithm for TCP retransmission timeouts. *SIGCOMM Comput. Commun. Rev. (CCR)*, 33(2):51–63, 2003.

[30] Y. Tian, K. Xu, and N. Ansari. TCP in wireless environments: problems and solutions. *IEEE Communications Magazine*, 43(3):S27–S32, 2005.

[31] 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 *ACM SIGCOMM Conference*, 2009.

[32] G. Wang and T. Ng. The impact of virtualization on network performance of Amazon EC2 data center. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2010.

[33] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: incast congestion control for TCP in data center networks. In *International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2010.

[34] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. R. Kompella, and D. Xu. vSlicer: latency-aware virtual machine scheduling via differentiated-frequency CPU slicing. In *ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2012.