Gatekeeper: Distributed Rate Control for Virtualized Datacenters

Paolo Victor Soares, Jose Renato Santos, Niraj Tolia, Dorgival Guedes

HP Laboratories
HPL-2010-151

Keyword(s):

Cloud Computing, Bandwidth, Virtualization, Virtual Machines, Rate Control, Virtual Switch

Abstract:

We present Gatekeeper, a distributed rate control mechanism that supports network link bandwidth guarantees for multiple co-located tenants in a virtualized datacenter. Gatekeeper provides network performance isolation across tenants by enforcing link bandwidth allocations for both egress and ingress network traffic at each physical host. Experiments on our Xen-based implementation of Gatekeeper in a datacenter cluster demonstrate effective control of link bandwidth for both TCP and UDP traffic, and for a Hadoop-based application running concurrently with streaming workloads.

External Posting Date: October 6, 2010 [Fulltext] Approved for External Publication
Internal Posting Date: October 6, 2010 [Fulltext]

Gatekeeper: Distributed Rate Control for Virtualized Datacenters

Paolo Victor Soares
UFMG, Brazil

Jose Renato Santos,
Yoshio Turner, Niraj Tolia
HP Labs, Palo Alto

Dorgival Guedes
UFMG, Brazil

ABSTRACT

We present Gatekeeper, a distributed rate control mechanism that supports network link bandwidth guarantees for multiple co-located tenants in a virtualized datacenter. Gatekeeper provides network performance isolation across tenants by enforcing link bandwidth allocations for both egress and ingress network traffic at each physical host. Experiments on our Xen-based implementation of Gatekeeper in a datacenter cluster demonstrate effective control of link bandwidth for both TCP and UDP traffic, and for a Hadoop-based application running concurrently with streaming workloads.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed Systems

General Terms

Distributed Cloud Bandwidth Control

Keywords

Cloud Computing, Bandwidth, Virtualization, Virtual Machines, Rate Control, Virtual Switch

1. INTRODUCTION

A typical datacenter hosts multiple services in a shared facility. The services are often consolidated onto physical servers using virtual machine (VM) technology [8, 3, 19]. Each service can consist of a collection of one or more VMs placed on one or more physical machines. With the rise of *cloud computing* [2], services will belong to mutually non-trusted *tenants* and exhibit varied and dynamic demands on datacenter resources.

These emerging environments will have a strong need for improved mechanisms to enforce performance isolation for tenants that share datacenter resources. While existing hypervisor mechanisms provide good support for allocating CPU and memory resources, only rudimentary support is currently available to manage the use of datacenter network I/O resources (for example, VMware ESX Server 3 can enforce parameter settings for average bandwidth, burst size, and peak bandwidth for each VM, but only in the transmit direction [10]). Effective management of network bandwidth will be crucial to handle the growing range of service workloads that stress local area network resources in the datacenter. For example, data-intensive applications on scalable frameworks like MapReduce [6] can be highly network-intensive. Also, future datacenters are expected to merge traditional messaging traffic with network storage traffic onto a single converged datacenter network fabric, using new network standards [5, 7] and new distributed storage and file systems [12].

The traditional performance bottleneck for datacenter local area networking was in the core of the fabric. Datacenter networks were constructed as tree topologies with high oversubscription ratios that severely limited network bisection bandwidth near the root of the tree. Thus, careful consideration of the entire network topology was needed to provide guaranteed bandwidth for each tenant. Recently, a flurry of advances in datacenter networking research [13, 21, 1, 20, 23], commercial products [11], and Ethernet standards [25] promises to make it practical to cost-effectively scale the bisection bandwidth of datacenter networks using multi-path switching. In addition, network topology-aware placement of service workloads in the datacenter can minimize the number of network hops that traffic traverses, reducing the aggregate load on network link and switch resources in the fabric core [16].

Our key observation is that the use of scalable datacenter networks shifts the bottleneck from the network fabric to the endpoint links that connect each physical server to the network fabric. This allows translating the problem of managing tenant network bandwidth into the (possibly more tractable) problem of managing each server's network *access links*. Thus, tenant bandwidth management can focus on the endpoint server links, which are potentially shared by all VMs hosted on a server, instead of having to reason about network bottlenecks that could arise anywhere in the fabric.

This paper presents Gatekeeper, a distributed rate control mechanism that provides network isolation for multi-tenant datacenters. Gatekeeper controls the usage of each server's network access link, which is presumed to be the bottleneck resource for datacenter network I/O. It provides per-VM link bandwidth guarantees in both directions of the link at each physical server, i.e., for both ingress and egress traffic. The logical view provided to a tenant is that all of its VMs attach to a single dedicated non-blocking switch, and each VM connects to the switch via an access link with a guaranteed level of bandwidth. Compared to static bandwidth allocations, the bandwidth guarantees in Gatekeeper provide more flexibility to achieve higher aggregate throughput. In particular, each VM can exceed its guaranteed allocation when extra bandwidth is available at both transmitting and receiving endpoints. Gatekeeper is a distributed mechanism that achieves scalability using a simple point-to-point protocol and minimal datacenter-wide control state – a small constant amount of state for each VM plus a small constant amount of state for each network access link.

Existing virtualization mechanisms can guarantee to each VM a specific fraction of the server's network link bandwidth, but only in the *transmit* direction. To our knowledge, no existing solution provides similar guarantees in the *receive* direction. Existing rate limiters in hosts and NICs are able to cap the maximum delivery rate to a VM by simply dropping any packets that are received in excess of the rate limits. Unfortunately, received packets only encounter the rate limiters after traversing the link. Hence, rate limiters alone are unable to control the use of the link bandwidth in the receive direction. Without link receive bandwidth guarantees, a service that receives a high rate of incoming network traffic can severely hurt the performance of co-located services. These ineffective rate limiters have the additional disadvantage that VMs are prevented from using bandwidth above their caps even when idle bandwidth is available.

Gatekeeper overcomes these limitations by approximating work-conserving scheduling while satisfying rate guarantees for all tenants that have traffic demands. It observes true traffic demands for each VM even though it encounters packets only after they traverse the physical link and uses that demand information to dynamically allocate bandwidth for each VM. To enforce these dynamic rate allocations, Gatekeeper takes advantage of the throughput adaptation property of TCP-friendly transport protocols. Specifically, Gatekeeper drops selected packets, causing remote senders to adjust transmission rates to satisfy the current allocation. For traffic that does not adjust rates in response to packet drops, Gatekeeper enforces dynamic rate assignments by imposing egress rate limits on remote senders through a distributed control protocol.

Gatekeeper is currently implemented in software at end hosts. This software-based approach enabled us to deploy and evaluate Gatekeeper on a real datacenter cluster. Alternatively, Gatekeeper could be implemented in hardware/firmware in host network interfaces (e.g., an Ethernet NIC) or edge switches. Our performance evaluation uses a mixture of streaming microbenchmarks and a Hadoop cluster application to show that Gatekeeper provides effective bandwidth guarantees that enable: 1) better control than with best-effort scheduling, and 2) better performance than using strict rate limits. A preliminary analysis of the CPU overhead of our current unoptimized implementation indicates that it uses only around 25% of a single CPU core to manage a gigabit Ethernet link.

2. GATEKEEPER ARCHITECTURE

A key design decision is to choose the form of network performance guarantees that Gatekeeper should provide to each tenant. The guarantee should be simple enough for users (customers) of a datacenter to reason about so that they can make appropriate requests to the datacenter provider when deploying tenants.

We chose to provide the tenant with the logical view that all VMs of the tenant are connected to a full crossbar switch. Each VM has an access link to the switch with a guaranteed bandwidth. However, a VM may find that it can exceed this bandwidth at times. As it is common in real physical deployments to attach multiple servers directly to the same switch, it is likely that our model will seem familiar to users who deploy tenants in a datacenter. Except for allowing bandwidth use above the guaranteed rate, this model is similar to the hose model [9, 13] in which throughputs are constrained only by the guaranteed bandwidths of the access links of the VMs.

The tenant model can be extended to include management of bandwidth between the tenant and external services or clients. For example, the logical full crossbar switch can include additional ports with assigned bandwidth for external client traffic. This requires Gatekeeper to operate on a router or gateway that handles all traffic exchanged between the external clients and the datacenter tenant. We note that this simple model could potentially be extended further to allow users to specify more general virtual topologies, but it is unclear whether many users would need that generality.

Figure 1 shows an overview of the Gatekeeper architecture. Gatekeeper intercepts packets sent from local VMs that are destined to traverse the server's network link, as well as packets received from the network link for local VMs. Internally, Gatekeeper has three components: the egress scheduler, the ingress scheduler, and the congestion agent.

The egress controller is depicted in Figure 2. The egress scheduler implements a traditional weighted fair scheduling policy with rate guarantees. The scheduler services packets to be transmitted from multiple queues.

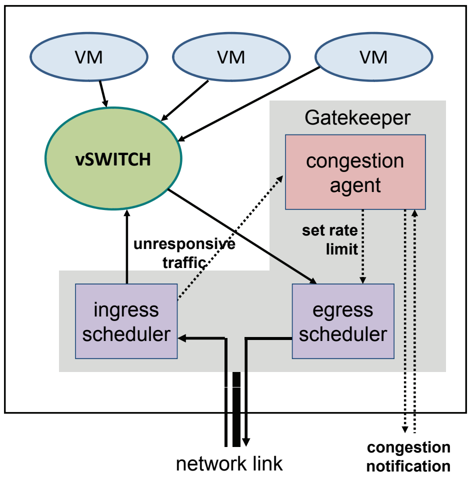


Figure 1: Architecture

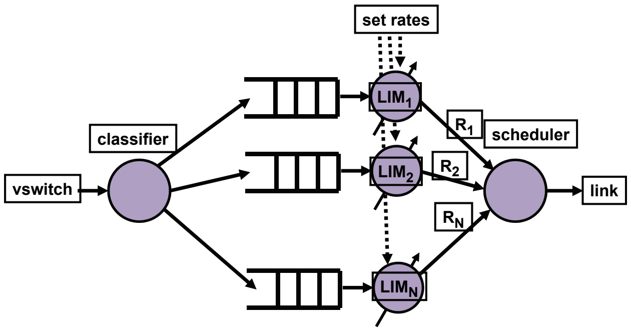


Figure 2: Egress Controller

Each queue is associated with a virtual machine, and when non-empty it is drained with at least its guaranteed link bandwidth rate R_i. Any excess bandwidth is distributed among queues that have additional traffic.

For the ingress scheduler, a simple approach would be to impose a rate limit that bounds the maximum rate at which each VM can receive traffic. However, this would amount to a static allocation of bandwidth and could lead to wasted bandwidth during periods when some VMs had demands for ingress traffic that were lower than their bandwidth allocations. During such periods, ideally another VM that had extra demands should be able to "borrow" the available bandwidth until the first VM experiences higher demand and reclaims its share. Thus, Gatekeeper seeks to achieve work-conserving scheduling for ingress traffic as well as for egress traffic.

To achieve work-conserving scheduling with guarantees, Gatekeeper must not only satisfy all guarantees, but also must satisfy demand in excess of guaranteed rates until either capacity or demand is exhausted. However, the challenge for ingress traffic is that the receiving host has incomplete knowledge of traffic demands.

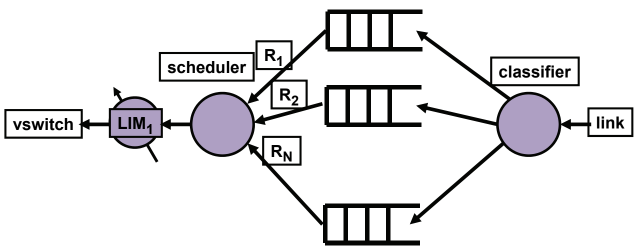


Figure 3: Ingress Controller

The receiving host only sees the arriving traffic that successfully traverses the physical link without getting dropped at the edge switch. For example, if the link is fully utilized, suppose one VM A is receiving at a rate higher than its guaranteed rate. If another co-located VM B is receiving at a rate lower than its guaranteed rate, it is not clear whether that is because there is no additional demand from VM B, or because there is additional demand which cannot be satisfied because the link is saturated receiving excess traffic for VM A. Gatekeeper uses two mechanisms working in concert to overcome this difficulty.

To expose the real traffic demand for each VM, Gatekeeper shifts the resource bottleneck for ingress traffic from the physical link to the Gatekeeper scheduler itself. As shown in Figure 3, the ingress controller places arriving packets into per-VM queues based on the destination MAC address, and delivers the traffic to the VMs using weighted fair queueing with rate guarantees. The ingress scheduler serves traffic to VMs at a maximum aggregate rate of LIM_{IN}, a rate which is slightly lower than the link rate. Maintaining this bandwidth headroom allows Gatekeeper to experience most traffic demands that would otherwise be shed at the edge switch. Conceptually, Gatekeeper uses this information to determine appropriate ingress bandwidth allocations at each moment in time for each VM to achieve approximately work-conserving scheduling with guarantees.

Instead of explicitly managing these dynamic bandwidth allocations for ingress traffic, Gatekeeper implicitly manages allocations by leveraging the throughput adaptation property of TCP-friendly transport protocols. When the aggregate arrival rate on the physical link exceeds LIM_{IN}, packets will accumulate in some queues of the ingress controller. Since the ingress scheduler services the queues with at least their guaranteed rate R_i, only the queues that are receiving packets at rates higher than their R_i can overflow and drop packets. For traffic that has good congestion control, the packet drops cause the remote senders to react by reducing transmission rates to satisfy the current implicit rate allocation (in addition, queuing alone helps because of the self-clocking behavior of TCP ACKs).

While dropping selected ingress packets causes TCP-friendly traffic to adapt to the available delivery rate, a remote VM may not reduce traffic rate appropriately in response to packet drops. For example, it can use non-TCP-friendly protocols like UDP, or a flood of very short-lived connections, or modified or misbehaving TCP implementations that disobey TCP's congestion control. To protect well-behaved TCP flows from such unresponsive traffic, Gatekeeper introduces a second mechanism which detects if the transmission rates fail to adjust to packet drops and, if so, imposes egress rate limits (LIM_j in Figure 2) on remote senders using a distributed control protocol. This assumes Gatekeeper is running on the sender side to react to the notification. Unresponsive traffic is detected by periodically monitoring packet drop rates at each ingress queue and generating a congestion notification message when the packet drop rate exceeds a threshold. This congestion message is sent to the sender side. Since many senders could be transmitting to a single receiver, a policy is needed to select which sender will be targeted for a congestion message. The policy used in the current version of Gatekeeper is to target the sender of the last packet that was dropped from the ingress queue. If multiple senders are transmitting to the same VM, it is possible that congestion messages are sent to VMs that are sending only TCP-friendly traffic in addition to VMs that are sending non-TCP-friendly traffic. We consider this acceptable since all senders belong to the same tenant.

Initially, the limit LIM_j is set equal to the physical link bandwidth and so does not impede transmission at all unless a congestion notification message for that VM is received. Gatekeeper at the sender side reacts to a congestion message by reducing the egress rate limit LIM_j for the sender VM that is the target of the congestion message. Gatekeeper reduces the VM's limit LIM_j by half on each congestion message. Then the rate is increased linearly over time until it reaches the maximum rate or another congestion notification is received. We experimentally determined the packet drop threshold to ensure that congestion notification messages are not triggered by TCP but instead only by unresponsive traffic. We adjusted the slope of the rate increase at the rate limiter to ensure that the distributed rate control favors well-behaved TCP connections.

To send congestion feedback messages back to the source, the current implementation of Gatekeeper maintains a directory service that maps the MAC addresses of all VMs to the address of the physical server that hosts each VM. Thus, Gatekeeper can identify the physical server of each transmitting VM and send congestion notifications to the Gatekeeper port on the sending physical servers. A possible alternative to this approach is to send congestion notifications to the MAC address of the transmitting VMs themselves but use a new ethertype. Gatekeeper could intercept all packets that use this new ethertype instead of delivering them to the VMs. While this would avoid the need for a directory service, we observe that equivalent directory services are already typically provided in systems that manage VMs, and could be leveraged by Gatekeeper.

Gatekeeper is currently implemented in host software (specifically, in a Xen dom0 driver domain) but could instead be implemented in NIC hardware or firmware. Alternatively, a slightly modified Gatekeeper design could be implemented in edge switches connected directly to servers. In that case, there is no need to limit the receive rate on the server network link to be less than the physical link bandwidth. However, since system management of VMs (e.g., VM live migration) needs to coordinate closely with the Gatekeeper mechanism, it seems preferred to implement Gatekeeper in servers rather than creating new dependencies between VM management and network switch management.

3. EVALUATION

Our Gatekeeper prototype based on the Xen hypervisor implements the congestion agent as a user-level process in Domain 0 (dom0) on each physical machine. The ingress and egress schedulers are implemented using the Linux traffic control framework TC in the dom0 kernel. The agent queries the schedulers using the rtnetfilter library to collect network statistics, such as per-VM bandwidth and packet drop rates. Each server in the test cluster has a Gigabit Ethernet link with maximum goodput of 941 Mb/s.

3.1 System Tuning

The Gatekeeper agent configures the TC mechanism with separate ingress and egress traffic queues for each virtual machine on each host. The agent periodically collects statistics for each queue, measuring the ingress arrival and drop rates. These measurements are taken at 10 msec intervals, and are averaged over the previous 100 msec to minimize noise. The ingress queue capacity determines the maximum traffic burst that can be received without packet dropping. This size should be sufficiently large to accommodate short term bursts caused by TCP's slow start and statistical interleaving of packets from multiple flows. However, the queue size should also be sufficiently short to enable fast reaction to changes in traffic demand.

We experimentally determined the minimum queue size that can accommodate a large number of simultaneous TCP flows without causing the packet drop rate at the ingress queue to exceed the ingress bandwidth headroom. As discussed in Section 2 this is needed to enable Gatekeeper to provide work-conserving scheduling for ingress traffic from the server side. We argue that providing a 5% headroom (reducing goodput from

941 Mb/s to ∼900 Mb/s on a gigabit link) is an acceptable price to pay for more predictable network performance and isolation. We determined the minimum ingress queue size that supports a large number of TCP connections while limiting the drop rate to at most half the ingress bandwidth headroom (20 Mb/s). To be conservative we assumed a worst case scenario where a large number of TCP flows start approximately simultaneously and enter slow start together. We run tests with a total of 256 simultaneous TCP connections to a single receiver. We varied the number of sender host servers from 1 to 16 while holding constant the total number of TCP connections. We found that a queue size of *160 packets* was sufficient to limit the drop rate to 20 Mb/s at all times during a 60 sec experiment. We also verified that the measured drop rate decreases as the number of sending hosts increases, and the highest drop rate is observed when all TCP connections originate at the same sender host. These results give us confidence that the selected queue size is large enough to absorb traffic bursts in most practical scenarios with well behaved TCP flows.

3.2 Evaluating Gatekeeper Behavior

Our initial evaluation uses a simple configuration with three servers hosting VMs from two customers, A and B, as shown in Figure 4. Host H1 runs one VM for each customer, while hosts H2 and H3 each run a single VM for customer A and customer B, respectively. Each customer runs a netperf microbenchmark between its two VMs. We examine two scenarios: 1) transmit (TX) bottleneck where traffic is transmitted from host H1 to the other corresponding server, and 2) receive (RX) bottleneck where traffic is transmitted from hosts H2 and H3 to the shared host H1. We evaluated different communication patterns varying the number of flows (none, 1, 10) and type of flow (TCP or UDP) for each customer. Due to a limitation of the current Linux TC implementation, the largest total rate limit below the link capacity that could be configured was 860 Mb/s. This reduces the maximum achievable bandwidth in these experiments, but we expect that the results would be similar if we could set a rate limit of 900 Mb/s (the 5% headroom would still be needed). To evaluate Gatekeeper's ability to allocate network bandwidth, we set targets of 75% of link bandwidth (645 Mb/s) for customer A and 25% (215 Mb/s) for customer B.

Figure 5 shows the rate achieved for each customer for different configurations. The x-axis shows the type and number of netperf flows for each customer. The horizontal dotted lines show the ideal rate allocated to the corresponding customer. Figure 5(a) shows that without Gatekeeper the rate achieved by one customer is significantly affected by the other customer's traffic. In particular, for the RX scenario a TCP connection

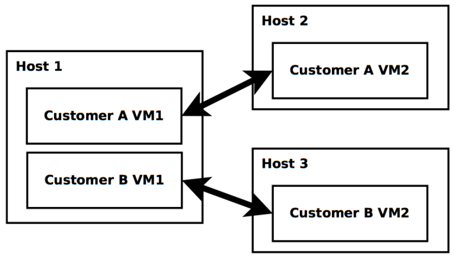


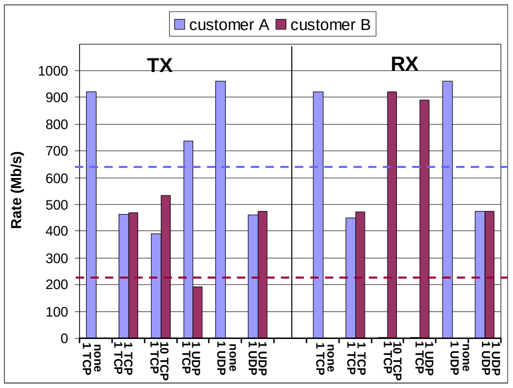
Figure 4: Experimental setup to evaluate Gatekeeper's basic functionality

from customer A is severely degraded when customer B has more TCP connections (10) or generates non-TCP-friendly network traffic such as UDP. This shows that traditional rate limiting at the transmitting side cannot provide network performance isolation. In contrast, Figure 5(b) shows that Gatekeeper enables customers with both UDP and TCP traffic to achieve rates close to their assigned bandwidths. At the same time, Gatekeeper detects any unused bandwidth by one customer and reassigns it to the other customer as shown in the experiments with no traffic from customer B.

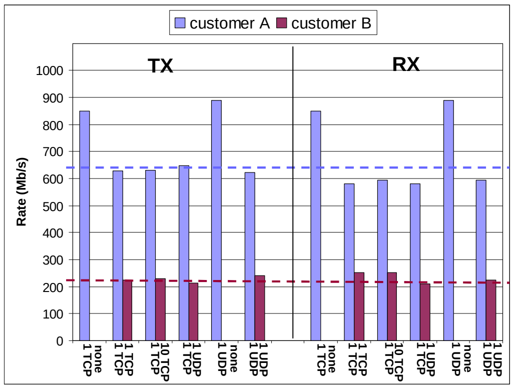
3.3 Hadoop Experiments

The previous results show that Gatekeeper successfully schedules network traffic to achieve the desired bandwidth allocation for a static scenario on a small number of nodes. We next evaluate Gatekeeper in a more realistic environment using a larger number of nodes and a workload with time-varying network demand. We consider a scenario with a Hadoop job running on 26 VMs distributed across 26 hosts and competing with background TCP and UDP traffic. The Hadoop job runs a custom application, named "Random", which in the Map phase generates random data lines from a set of input files and then in the Reduce phase counts the number of lines. The duration of the data transfer phase between the Map and Reduce phases is relatively long to ensure that the application performance is sensitive to the available network bandwidth. In addition, the sequence of Map, data transfer and Reduce phases exhibits a dynamic behavior with varying network demand over time.

The competing background TCP and UDP traffic is generated by netperf processes running on VMs co-located on the same 26 servers as the Hadoop job. These VMs were organized in 13 sender/receiver pairs. We evaluate two background traffic cases: one with 10 TCP streams and one with a UDP stream. We configured Gatekeeper with three different bandwidth allocations for the Hadoop job, 25% 50% and 75%, and allocated the remaining bandwidth to the background netperf traffic. Experiments were run 5 times and the results show a 95% confidence interval for each experiment.

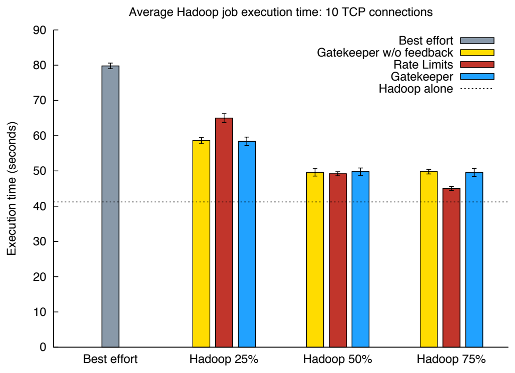


(a) Without Gatekeeper

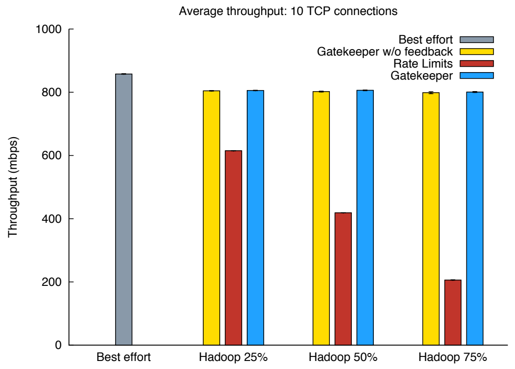


(b) With Gatekeeper

Figure 5: Results for simple scenarios



(a) Hadoop execution time



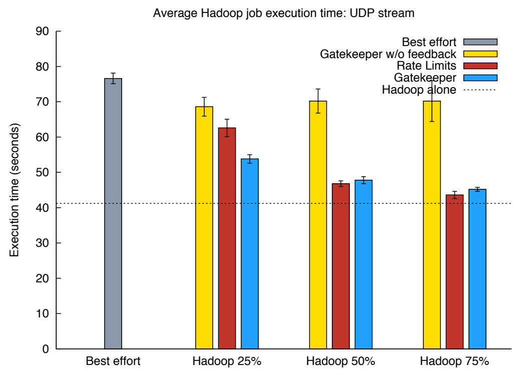
(b) TCP streams performance

Figure 6: Hadoop job competing with 10 TCP streams

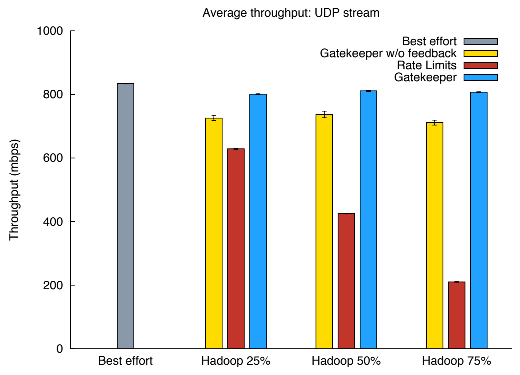
The results for background traffic with 10 TCP connections in each sender/receiver pair are shown in Figure 6. Figure 6(a) shows the Hadoop job execution time for different scheduling mechanisms and different Hadoop bandwidth allocations, and Figure 6(b) shows the average throughput of netperf sender/receiver pairs. The rate limit case shows the results for a simple static rate limiter for both the ingress and egress traffic. For Gatekeeper, results are shown with and without the congestion feedback mechanism used to detect unresponsive traffic. Finally, the best effort case shows performance with no rate control. The dotted line shows the optimal performance that is achieved by the Hadoop job when there is no competing background traffic. When no rate control is used, the 10 TCP connections significantly affect the performance of the Hadoop job causing the execution time to increase from 41 to 75 seconds (more than 70% slowdown). When rate control is used, Hadoop execution time decreases as expected as the bandwidth allocated to Hadoop is increased. In general, Hadoop performance under Gatekeeper is similar to a simple rate limiter mechanism, but Gatekeeper detects bandwidth unused by Hadoop during the Map and Reduce phase and re-assigns it to the netperf flows, as shown in Figure 6(b). Gatekeeper provides much higher bandwidth to the netperf flows while providing similar performance for Hadoop. Since TCP is well behaved, Gatekeeper avoids using the congestion control feedback mechanism, and thus the two Gatekeeper cases show identical performance.

While Gatekeeper performs well in most scenarios, it provides slightly lower performance (8%) than with static rate limits at high bandwidth allocations (75%) In this case, Gatekeeper's drop policy at the receiver nodes ends up causing slightly more drops than static rate limits because of the higher rate of the competing traffic. However, when the bandwidth allocation for Hadoop is lower (25%), Gatekeeper outperforms the static allocation. This is a consequence of using a work conserving scheme in Gatekeeper. In our scenario netperf streams generate egress traffic at only half of the nodes and receive ingress traffic at the other half. Gatekeeper detects this unused bandwidth and re-assigns it to the Hadoop job, improving its performance.

Figure 7 shows that Gatekeeper's congestion feedback mechanism maintains good performance isolation in the presence of competing unresponsive UDP traffic, providing Hadoop performance equivalent to that achieved



(a) Hadoop execution time



(b) UDP streams performance

Figure 7: Hadoop job competing with UDP stream

with static rate limits while providing better use of the available bandwidth by the netperf traffic.

4. RELATED WORK

Bandwidth controls in modern NICs, operating systems and hypervisors merely cap maximum delivery rates to VMs. Thus, these solutions cannot match the ability of Gatekeeper to provide link bandwidth guarantees. Moreover, rate caps are non-work-conserving and thus often waste idle bandwidth.

Several recent efforts propose to rearchitect Layer-2 data center networks to provide scalable bisection bandwidth [1, 13, 21, 20, 23]. Gatekeeper assumes that fabric bandwidth is well provisioned and focuses on managing the links that provide a server access to the network fabric. Each access link can be shared by VMs belonging to different services. Thus, Gatekeeper extends network bandwidth guarantees from the network edge all the way to end-point VMs.

More closely related work is Core-Stateless Fair Queueing (CSFQ) [24] for the Internet. In CFSQ, all state is maintained by the edge routers but packet labels allow the core to drop traffic and thus achieves approximately fair bandwidth allocation. CSFQ assumes the end-nodes cannot be trusted and therefore concentrates all trust in the core. In contrast, Gatekeeper concentrates on a managed cloud environment and does not need to rely on switch or router support.

Gatekeeper also differs significantly from other router-based mechanisms to provide QoS and prevent Distributed Denial-of-Service attacks [15]. In particular, Gatekeeper is complementary to Cloud Control [22]. While Cloud Control performs distributed rate limiting, it uses a complex distributed protocol to enforce traffic constraints between different data center locations. Gatekeeper, in comparison, focuses on providing distributed rate guarantees within a single data center and uses a simpler approach to meet its goals.

The congestion feedback mechanism of Gatekeeper may be conceptually related to the in-progress IEEE 802.1Qau standards effort to extend Ethernet to support some type of congestion notification. However, this standard is not yet released, and Gatekeeper congestion feedback works with unmodified Ethernet.

OpenFlow [18] is gaining traction in providing programmable network switches that can provide differentiated routing or other treatment depending on a flexible match of field values in packet headers. Several projects (e.g., SANE [17], Ethane [4], NOX [14]) leverage OpenFlow to provide centralized network-wide management of performance, security, and other networking properties. This work is complementary to Gatekeeper. Individual flows or traffic classes used by VMs could be managed in the fabric using centralized network management of NOX or similar systems, while Gatekeeper provides bandwidth guarantees for VMs at the endpoint links.

Finally, VM placement and reallocation systems migrate VMs based on the load at each physical machine, mainly in terms of CPU usage [26, 27]. While migration could also be triggered by insufficient network bandwidth, Gatekeeper can help these systems guarantee network bandwidth and reduce the need for migration.

5. CONCLUSION

In this paper we showed that traffic isolation between virtualized datacenter tenants is achievable by combining egress and ingress traffic control at the end machines using drops and explicit feedback. We presented Gatekeeper, our Xen-based prototype, and described how it was tuned to perform under different load conditions with acceptable drop rates. Our evaluation showed that Gatekeeper is a viable solution for achieving good rate guarantees while allowing efficient use of unused reserved bandwidth. In tests combining a Hadoop workload and a network-intensive application, Gatekeeper was able to preserve good execution times for Hadoop while allowing the network-intensive workload to use all available bandwidth in periods when the Hadoop application was not using the network. As expected, ingress control and drops were sufficient to reach good results for TCP-based traffic, while the feedback mechanism used to limit egress traffic was essential when UDP was present. In future work we intend to extend the control mechanisms to refine the control of TCP traffic and to take into account complex network topologies. We also plan to combine Gatekeeper with VM migration management so that VM placement can factor in observed traffic behavior and reservations.

6. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference*, pages 63–74, Seattle, WA, Aug. 2008.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb. 2009.

[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 *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[4] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: taking control of the enterprise. In *Proceedings of the ACM SIGCOMM 2007 Conference*, pages 1–12, Kyoto, Japan, Aug. 2007.

[5] D. Cohen, T. Talpey, A. Kanevsky, U. Cummings, M. Krause, R. Recio, D. Crupnicoff, L. Dickman, and P. Grun. Remote direct memory access over the converged enhanced ethernet fabric: Evaluating the options. In *Proceedings of the 2009 17th IEEE Symposium on High Performance Interconnects (HOTI '09)*, New York, NY, Aug. 2009.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI '04)*, pages 137–150, San Francisco, CA, Dec. 2004.

[7] C. DeSanti and J. Jiang. Fcoe in perspective. In *Proceedings of the 2008 International Conference on Advanced Infocomm Technology (ICAIT '08)*, pages 1–8, Shenzhen, China, July 2008.

[8] S. Devine, E. Bugnion, and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. VMware US Patent 6397242, Oct 1998.

[9] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive. A flexible model for resource management in virtual private networks. In *Proceedings of the ACM SIGCOMM 1999 Conference*, pages 95–108, New York, NY, USA, 1999. ACM.

[10] Jan. 2010. `http://www.vmware.com/pdf/vi3_35/esx_3/r35/vi3_35_25_3_server_config.pdf`.

[11] Fulcrum Microsystems. FocalPoint in large-scale Clos switches, Oct. 2007.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–43, Bolton Landing, NY, Oct. 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 *Proceedings of the ACM SIGCOMM 2009 Conference*, pages 51–62, Barcelona, Spain, Aug. 2009.

[14] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *Computer Communication Review*, 38(3):105–110, 2008.

[15] J. Ioannidis and S. M. Bellovin. Implementing pushback: Router-based defense against ddos attacks. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2002.

[16] G. Lee, N. Tolia, P. Ranganathan, and R. H. Katz. A case for topology-aware resource allocation for data-intensive applications in the cloud. Technical Report HPL-2009-335, HP Labs, Palo Alto, CA, 2009.

[17] J. Luo, J. Pettit, M. Casado, J. Lockwood, and N. McKeown. Prototyping fast, simple, secure switches for ethane. In *Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects (HOTI '07)*, pages 73–82, Palo Alto, CA, Aug. 2007.

[18] N. McKeown, T. Anderson, H. Balakrishnan, G. M. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, and J. S. Turner. Openflow: enabling innovation in campus networks. *Computer Communication Review*, 38(2):69–74, 2008.

[19] Jan. 2010. `http://www.microsoft.com/hyper-v-server`.

[20] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. Spain: Cots data-center ethernet for multipathing over arbitrary topologies. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, Apr. 2010.

[21] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, S. R. Pardis Miri, V. Subramanya, and A. Vahdat. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 Conference*, Barcelona, Spain, Aug. 2009.

[22] B. Raghavan, K. V. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren. Cloud control with distributed rate limiting. In *Proceedings of the ACM SIGCOMM 2007 Conference*, pages 337–348, Kyoto, Japan, Aug. 2007.

[23] M. Schlansker, J. Tourrilhes, Y. Turner, and J. R. Santos. Killer fabrics for scalable datacenters. In *IEEE International Conference on Communications (ICC)*, May 2010.

[24] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: a scalable architecture to approximate fair bandwidth allocations in high-speed networks. *IEEE/ACM Transactions on Networks*, 11(1):33–46, 2003.

[25] J. Touch. RFC 5556: Transparent interconnection of lots of links (trill): Problem and applicability statement, May 2009.

[26] Dynamic balancing and allocation of resources for virtual machines, 2008. `http://www.vmware.com/files/pdf/drs_datasheet.pdf`.

[27] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 229–242, Apr. 2007.