Improving Network Performance for Cloud-hosted Services

Thesis proposal

Keqiang He
Department of Computer Science
University of Wisconsin-Madison
keqhe@cs.wisc.edu

Wednesday 10th August, 2016

Abstract

Cloud computing has been shown to be a great success and it continues to grow rapidly. Networking is a critical component for high performance cloud computing infrastructures because services and tasks running in datacenters can be affected significantly due to poor network performance. Therefore, understanding and improving network performance for cloud-hosted services is an important problem.

This proposal explores how to improve network performance for cloud-hosted services from different perspectives—traffic load balancing, network congestion control, datacenter network architecture design and web service deployment. First, we designed and implemented a software edge-based load balancing scheme for high speed (10G+) datacenter networks that can achieve near-perfect traffic load balancing. Second, we characterized how today's web services are using the public clouds and shed light on how to improve quality of service for web services. Finally, we propose two research topics we plan to explore—1) how to perform congestion control enforcement for public data centers where tenants' VMs can have out-dated, inefficient, or misconfigured TCP stacks and 2) build a network architecture analysis framework to systematically quantify different network architectures and compare them based on cost, wiring complexity, bisection bandwidth, fault-tolerance and similar metrics. Furthermore, we plan to use this systematic approach to identify novel datacenter network architectures.

Contents

1 Introduction

1.1 Background and Motivation

Cloud computing is changing the way computing is conducted since a few years ago. It is a rapidly growing business and many industry leaders (e.g., Amazon [14], Microsoft [61], IBM [43, 44] and Google [36]) have embraced such a business model and are deploying highly advanced cloud computing infrastructures. Market analysis [25] has predicted that the global cloud computing market will reach $270 billion by 2020. The success of cloud computing is not accidental—it is rooted in many advantages that cloud computing offers over traditional computing model. The most notable feature is that tenants (customers) who rent the computing resources can get equivalent computing power with *lower cost*. That is because the computing resources (CPUs, memory, storage, and network) are shared among multiple users and server consolidation and server virtualization improves the utilization of the computing resources. Another key advantage cloud computing offers is *computing agility*. That means, tenants can rent as many computing resources as they needed and can grow or shrink the computing pool based on their demands. This feature is especially attractive for relatively smaller and rapidly growing businesses.

Datacenter network is a key enabler for high performance cloud computing infrastructures because network performance can affect services and tasks running in datacenters significantly. Therefore, understanding and improving network performance for cloud-hosted services is an important and timely research direction. In this proposal, we will research solutions that help improve cloud network performance.

1.2 Research Questions

This research exams some of the most important components that affect cloud network performance. These components include traffic load balancing, congestion control, web server deployment and datacenter network architecture design. We study the current practices for each component and explore the possibility to improve them. More specifically, we try to answer the following questions:

1. How can we load balance the traffic within the datacenter to minimize the likelihood of network congestion? If we can (largely) eliminate network congestion, then cloud-hosted services' performance can be improved significantly.

2. Who are the existing users in public clouds? How they are using the cloud? What are the deployment patterns of current web services and are they optimal? How we can improve cloud-hosted web services' availability and wide area network performance?

3. In multi-tenant cloud where tenant VMs can use out-dated, inefficient, or misconfigured TCP stacks, is it possible for administrators to take control of a VM's TCP congestion control algorithm *without* making changes to the VM or network hardware to reduce network queueing latency and improve throughput fairness?

4. Can we build a datacenter network architecture analysis framework to compare existing datacenter network architectures based on metrics such as wiring complexity, bandwidth, fault-tolerance and routing convergence? Can we use this analysis framework to explore new

datacenter network topologies and codesign routing, load balancing to improve robustness and performance?

In what follows, we will present how we answer these research questions. We first introduce works that are completed (questions 1, 2). Then we will discuss some new topics we plan to explore (in questions 3 and 4).

1.3 Completed Work

1.3.1 Software edge-based load balancing scheme for fast datacenter networks

Datacenter networks must support an increasingly diverse set of workloads. Small latency-sensitive flows to support real-time applications such as search, RPCs, or gaming share the network with large throughput-sensitive flows for video, big data analytics, or VM migration. Load balancing the network is crucial to ensure operational efficiency and suitable application performance. Unfortunately, popular flow-hashing-based load balancing schemes, e.g., ECMP, cause congestion when hash collisions occur [6, 21, 28, 76, 77, 87] and perform poorly in asymmetric topologies [9, 89]. Conceptually, ECMP's flaws are not internal to its operation but are caused by asymmetry in network topology (or capacities) and variation in flow sizes. *In a symmetric network topology where all flows are "mice", ECMP should provide near optimal load balancing*; indeed, prior work [9, 82] has shown the traffic imbalance ECMP imposes across links goes down with an increase in the number of flows and a reduction in the variance of the flow size distribution.

We leverage this insight to design a high performance proactive load balancing scheme without requiring special purpose hardware or modifications to end-point transport. It relies on the datacenter network's *software edge* to transform arbitrary sized flows into a large number of near uniformly sized small sub-flows, and to proactively spread those uniform data units over the network in a balanced fashion. The proposed scheme is fast (works at 10+ Gbps), and doesn't require network stack configurations that may not be widely supported outside the datacenter (such as increasing MTU sizes). It piggybacks on recent trends where several network functions, e.g., firewalls and application-level load balancers, are moving into hypervisors and software virtual switches on end-hosts [50, 72, 73]. Several challenges arise when employing the edge to load balance the network on a sub-flow level. Software is slower than hardware, so operating at 10+ Gbps speeds means algorithms must be simple, light-weight, and take advantage of optimizations in the networking stack and offload features in the NIC. Any sub-flow level load balancing should also be robust against reordering. Reordering not only impacts TCP's congestion control mechanism, but also imposes significant computational strain on hosts, effectively limiting TCP's achievable bandwidth if not properly controlled (see §3.2). Last, the approach must be resilient to hardware or link failures and be adaptive to network asymmetry.

We build a proactive load balancing system called Presto to solve the challenges mentioned above. Presto utilizes vSwitches to break flows into discrete units of packets, called *flowcells*, and distributes them evenly to near-optimally load balance the network. Presto uses the maximum TCP Segment Offload (TSO) size (64 KB) as flowcell granularity, allowing for fine-grained load balancing at network speeds of 10+ Gbps. To combat reordering, we modify the Generic Receive Offload (GRO) handler in the hypervisor OS to mitigate the computational burden imposed by reordering and prevent reordered packets from being pushed up the networking stack. Finally, we show Presto can load balance the network in the face of asymmetry and failures using a combination of fast failover and weighted multipathing at the network edge. We evaluate Presto on a real 10 Gbps testbed. Our experiments show Presto outperforms existing load balancing schemes (including flowlet switching, ECMP, MPTCP) and is able to track the performance of a single, non-blocking switch (an optimal case) within a few percentage points over a variety of workloads, including trace-driven. Presto improves throughput, latency and fairness in the network and also reduces the flow completion time tail for mice flows. This work is published in ACM SIGCOMM'15 [42].

1.3.2 Characterizing web service deployment in public clouds

Today, web services are increasingly being deployed in infrastructure-as-a-service (IaaS) clouds such as Amazon EC2, Windows Azure, IBM SoftLayer and Rackspace. Industry and the media claim that over 1% of Internet traffic goes to EC2 [59] and that outages in EC2 are reputed to hamper a huge variety of services [31, 32, 33, 80]. Despite the popularity of public IaaS clouds, we are unaware of any in-depth measurement study exploring the current usage patterns of these environments. Prior measurement studies have either quantified the compute, storage, and network performance these clouds deliver [55, 56], evaluated the performance and usage patterns of specific services that are hosted in these clouds, e.g., Dropbox [29], or examined cloud usage solely in terms of traffic volume [52].

We present the first in-depth empirical study of modern IaaS clouds that examines *IaaS cloud usage patterns* and identifies *ways in which cloud tenants could better leverage IaaS clouds to improve performance*. Our focus is particularly on web services hosted within IaaS clouds, which our study (unsurprisingly) indicates is a large and important usage case for IaaS. We first examine *who is using public IaaS clouds*. We generate a dataset of cloud-using domains using extensive DNS probing in order to compare the IPs associated with websites on Alexa's top 1 million list [7] against published lists of cloud IP ranges. This identifies that $\approx$40K popular domains (4% of the Alexa top million) have a subdomain running atop Amazon EC2 or Windows Azure, two of the largest public clouds. We extract an additional $\approx$13K cloud-using domains from a full packet capture taken at the edge of our university, and use this capture to characterize the network traffic patterns of cloud-hosted web services. These results indicate that a large fraction of important web services are already hosted within public IaaS clouds.

Further, we dissect *how these services are using the cloud*. EC2 and Azure both have a veritable potpourri of features, including virtual machines, load balancers, platform-as-a-service (PaaS) environments, content-distribution networks (CDNs), and domain name services. They also give tenants the choice of deploying their services in several different regions (i.e., geographically distinct data centers), and EC2 provides several different "availability zones" within each region. We couple analysis of DNS records with two different cloud cartography techniques [79] to identify which features, regions and zones web services use. We identify several common front end deployment patterns and report estimates of the percentages of Alexa subdomains using each of the patterns. In particular, we find that about 4% of EC2-using web services use load balancers and 8% of them make use of PaaS. We also show that 97% of subdomains hosted on EC2 and 92% of subdomains hosted on Azure are deployed in only a single region. Counted among these are the subdomains of most of the top 10 (by Alexa rank) cloud-using domains. Services deployed in EC2 also appear to make limited use of different availability zones: our measurements estimate that only 66% of subdomains use more than one zone and only 22% use more than two. This lack of redundancy means that many (even highly ranked Alexa) services will not tolerate single-region or even single-zone failures.

Finally, we use a series of PlanetLab-based [74] active measurements and simulations to *estimate the impact of wide-area route outages and the potential for wide-area performance improvement*. We find that expanding a deployment from one region to three could yield 33% lower average latency experienced by globally distributed clients, while also substantially reducing the risk of service downtime due to downstream Internet routing failures. This work is published in ACM IMC'13 [41].

1.4 Proposed Work

1.4.1 Congestion control enforcement for improved datacenter networks

Multi-tenant datacenters are successful because tenants can seamlessly port their workloads, applications and services to the cloud. Virtual Machine (VM) technology plays an integral role in this success by enabling a diverse set of operating systems and software to be run on a unified underlying framework. This flexibility, however, comes at the cost of dealing with out-dated, inefficient, or misconfigured TCP stacks implemented in the VMs. We investigate if administrators can take control of a VM's TCP congestion control algorithm *without* making changes to the VM or network hardware. We propose a virtual switch-based congestion control enforcement scheme that exerts fine-grained control over arbitrary tenant TCP stacks by enforcing per-flow congestion control in the vSwitch. Our scheme is light-weight, flexible, scalable and can police non-conforming flows.

1.4.2 Datacenter network architecture analysis and exploration

Although there have been many datacenter network topologies proposed [5, 37, 66, 81], to date, there are very few datacenter architecture analysis frameworks that help network architect evaluate different network topologies at large scale. Hence, we propose to build a network architecture analysis framework that helps compare different network architectures based on bisection bandwidth, latency, fault tolerance and routing convergence time and similar metrics. Based on this analysis framework, we will investigate if we can identify new topologies that can improve the current practices and codesign routing, load balancing and network topology.

1.5 Structure of Thesis Proposal

The rest of this proposal is organized as follows: Section 2 discusses related work. Section 3 discusses the edge-based traffic load balancing scheme for datacenter networks. Section 4 discusses our work on characterizing web service deployment in public cloud, our observations and suggestions to improve network performance and robustness. Section 5 presents two new research topics I plan to work on and gives an estimated timeline to finish the rest part of my thesis.

Section 3 and 4 are based on previous publications [41, 42].

2 Related Work

We summarize the related work on improving data center/cloud network's *performance* below. We focus on the following categories of works: 1)Datacenter network architectures to provide massive bandwidth and strong fault-tolerance, 2)traffic engineering and load balancing within the data center to meet throughput and latency demands of the cloud-hosted workloads, 3)congestion control and transport protocols to mitigate the extent of network congestion and reduce network latency and 4)improving end-users' Internet access to cloud-hosted services.

Datacenter Network Architectures Many datacenter network architectures have been proposed. The seminal work FatTree [5] (FatTree is a 3-stage Clos network [24]) proposes to use cheap Ethernet switches to construct FatTree topologies to reduce cost and supports tens of thousands of servers with full bisection bandwidth. For example, using 48-port GigE Ethernet switches, FatTree topology can support up to 27,648 servers and there are 576 equal-cost paths between any given hosts pair that are not located in the same pod. ECMP can be employed to spread flows across multiple links to achieve load balancing. Because of massive multipathing, FatTree topology's bandwidth degrades gracefully when link failure happens. Similarly, VL2 [37] is also a 3-stage Clos topology, the difference from FatTree is that VL2 only uses 1Gbps links between ToR (Top-of-Rack) switches and servers and uses 10Gbps elsewhere. This reduces wiring complexity of the topology. VL2 uses centralized ARP and link-state routing protocol. VL2 proposes to decouple locator IP address and application IP address to provide virtual layer 2 networks using IP-in-IP encapsulation which inspired the development of VXLAN [58]. Portland [66] tries to provide layer 2 semantics (i.e., an VM can has any IP address, regardless of its location), so it uses pseudo MAC address to forward packets and when packets arrive at the destination edge switch pseudo MAC is rewritten to actual MAC. Pseudo MACs are assigned based on the location of the edge switch (the goal of positional pseudo MAC is to use wildcard rules to reduce the number of rules needed in the switch). A fabric manager is used to perform ARP and failure handling. When failure occurs, location discovery message exchanged between switches will discover link failures and report to the fabric manager, then the fabric manager updates forwarding rules in the affected switches to bypass the failed path. The topology is FatTree (3-stage Clos network) and all the links have the same speed (1Gbps). Multipathing is assumed to be achieved by ECMP. Jupiter Rising [81] presents the evolution of Google's datacenter networks in the last decade. The key ideas are: using multi-stage Clos topology to scale out, using merchant chips to build switching blocks to reduce cost and using centralized routing protocol to manage the routing behavior. Routing protocol (i.e., firepath) is centralized and upon failures, neighbor discovery messages exchanged between switches will identify link down event and firepath will push the topology state changes to switches. Then firepath agents on each switch individually re-calculate the forwarding states to bypass failed links. Different from Portland, in firepath, topology state is managed centralized but routing calculation is distributed. The routing convergence time after link failure is hundreds of ms to several seconds. Multipathing is achieved by ECMP.

Traffic Engineering and Load Balancing in Datacenters MPTCP [76] is a transport protocol that uses subflows to transmit over multiple paths. Transport layer solutions such as MPTCP require widespread adoption and are difficult to enforce in multi-tenant datacenters where customers often deploy customized virtual machines (VMs). CONGA [9] and Juniper VCF [40] employ congestion-aware flowlet switching [82] on specialized switch chipsets to effectively load balance the network. CONGA [9] and Juniper VCF [40] need to replace the edge switches in the data center. RPS [28] and DRB [21] perform per-packet load balancing on symmetric 1 Gbps networks at the switch and end-host, respectively. Per-packet load balancing can incur significant end-host overhead for DRB if not resorting to jumbo frames. Packet reordering problem is not well considered in both RPS and DRB. RPS and DRB may perform worse than ECMP in case of topology asymmetry. Hedera [6], MicroTE [17] and Planck [77] use centralized traffic engineering to reroute traffic based on network conditions. Fastpass [71] employs a centralized arbiter to schedule path selection for each packet. Centralized schemes are fundamentally reactive to congestion, and are very coarse-grained due to the large time constraints of their control loops [6], require extra network infrastructure [77] or suffer from a single point of failure [71]. FlowBender [46] reroutes flows when congestion is detected by end-hosts but network congestion already happened before FlowBender can react to it.

Congestion Control and Transport Protocols in Datacenters DeTail [87] is a cross-layer network stack designed to reduce the tail of flow completion times. DCTCP [10] is a transport protocol that uses the portion of marked packets by ECN to adaptively adjust TCP's congestion window to reduce switch buffer occupancy. Thus, DCTCP can reduce flow completion time. HULL [11] uses Phantom Queues and congestion notifications to cap link utilization and prevent congestion. [85] advocates to reduce TCP $\mathrm{RTO}_{\mathrm{min}}$ (200 ms in default in Linux) value to fine-grained values (sub-millisecond level) to solve the TCP incast problem [23]. ICTCP [86] proposes to monitor TCP throughputs at the receiver side and adaptively adjust the TCP receiver window size to reduce TCP throughput and avoid packet drops caused by TCP incast congestion. TCP-Bolt [83] identifies three serious performance issues in lossless Ethernet: large buffering delays, TCP throughput unfairness and head-of-line blocking. The authors of TCP-Bolt propose a variant of DCTCP (DCTCP without slow start to best utilize the benefits of lossless Ethernet) to solve the three key performance issues. DCQCN [90] also attacks the large buffering delay, TCP unfairness and head-of-line blocking problems in lossless Ethernet, but DCQCN works for a slightly different scenario: Remote Direct Memory Access (RDMA) over drop-free IP-routed networks and 40Gbps line-rate for highly demanding application. Similar to TCP-Bolt, DCQCN applies DCTCP-like congestion control and switch marking to lossless network to solve the three above mentioned performance issues. DCQCN is implemented on Mellanox NICs to reduce CPU overhead to meet the high speed (40G) requirement. TIMELY [62] and DX [53] revisit the possibility of using RTT to perform congestion control for data center networks. The conventional wisdom is that RTT-based congestion control is not suitable for data center networks because of its low latency characteristic. However, with modern network interface cards (NICs), it is possible to use hardware timestamps provided by the NIC to get accurate RTT measurement to perform congestion control for the data centers.

Improving End-users' Internet Access to Cloud-hosted Services Reducing the latency for Internet end-users' access to the cloud-hosted services is critical to revenue. [35] observes that TCP's timeout-driven recovery causes web transfers take 5 time longer than average. Based on this observation, it proposes three kinds of loss recovery schemes to reduce the web access latency. Their experiments show that they can cut the average latency by 23% and 99th percentile latency by 47%. Ananta [68] proposes a software-based load balancer to load balance the north-south traffic (i.e., the traffic between Internet users and the services in the cloud) such that the users requests can be distributed properly among a large set of servers and user access latency is reduced. Amazon Route 53 [13] is a DNS system and it can direct the end-users requests to the services running in Amazon AWS. Route 53 has the ability to direct the requests to the servers with the least latency to the end-users so it can help improve user experience.

3 Presto: Edge-based Load Balancing for Fast Datacenter Networks

3.1 Introduction

Presto is a new load balancing scheme for data center networks. It utilizes the software edge (virtual switch in the hypervisor) and TCP offloading features (TCP Segmentation Offload and Generic Receive Offload) to achieve near-perfect traffic load balancing on symmetric networks (e.g., 2-tier Clos) with very little CPU overhead. This work makes the following contributions:

1. We design and implement a system, called Presto, that near-optimally load balances links in the network. We show that such a system can be built with no changes to the transport layer or network hardware, and scales to 10+ Gbps networking speeds. Our approach makes judicious use of middleware already implemented in most hypervisors today: Open vSwitch and the TCP receive offload engine in the OS (Generic Receive Offload, GRO, in the Linux kernel). [1]

2. We uncover the importance of GRO on performance when packets are reordered. At network speeds of 10+ Gbps, current GRO algorithms are unable to sustain line rate under severe reordering due to extreme computational overhead, and hence per-packet load-balancing approaches [21, 28] need to be reconsidered. We improve GRO to prevent reordering while ensuring computational overhead is limited. We argue GRO is the most natural place to handle reordering because it can mask reordering in a light-weight manner while simultaneously limiting CPU overhead by having a direct impact on the segment sizes pushed up the networking stack.

3. Presto achieves near-optimal load balancing in a proactive manner. For that, it leverages symmetry in the network topology to ensure that all paths between a pair of hosts are equally congested. However, asymmetries can arise due to failures. We demonstrate Presto can recover from network failures and adapt to asymmetric network topologies using a combination of fast failover and weighted multipathing at the network edge.

4. Finally, we evaluate Presto on a real 10 Gbps testbed. Our experiments show Presto outperforms existing load balancing schemes (including flowlet switching, ECMP, MPTCP) and is able to track the performance of a single, non-blocking switch (an optimal case) within a few percentage points over a variety of workloads, including trace-driven. Presto improves throughput, latency and fairness in the network and also reduces the flow completion time tail for mice flows.

3.2 Design Decisions and Challenges

In Presto, we make several design choices to build a highly robust and scalable system that provides near optimal load balancing without requiring changes to the transport layer or switch hardware. We now discuss our design decisions.

[1] Also known as Receive Segment Coalescing (RSC) [78], or in hardware, Large Receive Offload (LRO) [38]

3.2.1 Design Decisions

Load Balancing in the Soft Edge A key design decision in Presto is to implement the functionality in the soft edge (i.e., the vSwitch and hypervisor) of the network. The vSwitch occupies a unique position in the networking stack in that it can easily modify packets without requiring any changes to customer VMs or transport layers. Functionality built into the vSwitch can be made aware of the underlying hardware offload features presented by the NIC and OS, meaning it can be fast. Furthermore, an open, software-based approach prevents extra hardware cost and vendor lock-in, and allows for simplified network management. These criteria are important for providers today [64]. Thanks to projects like Open vSwitch, soft-switching platforms are now fast, mature, open source, adopted widely, remotely configurable, SDN-enabled, and feature-rich [50, 72]. Presto is built on these platforms.

Reactive vs Proactive Load Balancing The second major design decision in Presto is to use a proactive approach to congestion management. Bursty behavior in datacenter workloads can create transient congestion issues that must be reacted to before switch buffers overflow to prevent loss (timescales range from hundreds of microseconds to 4 ms [77]). This requirement renders most of the centralized reactive schemes ineffective as they are often too slow to react to any but the largest network events, e.g., link failures. Furthermore, centralized schemes can hurt performance when rerouting flows using stale information. Distributed reactive schemes like MPTCP [76] and CONGA [9] can respond to congestion at faster timescales, but have a high barrier to deployment. Furthermore, distributed reactive schemes must take great care to avoid oscillations. Presto takes a proactive, correct-by-design approach to congestion management. That is, if small, uniform portions of traffic are equally balanced over a symmetric network topology, then we don't need to be reactive to congestion. Presto is only reactive to network events such as link failures. Fortunately, the higher timescales of the reactive feedback loops are sufficient in these scenarios.

Load Balancing Granularity ECMP has been shown to be ineffective at load balancing the network, and thus many schemes advocate load balancing at a finer granularity than a flow [9, 21, 28, 40]. A key factor impacting the choice of granularity is operating at high speed. Operating at 10+ Gbps incurs great computational overhead, and therefore host-based load balancing schemes must be fast, light-weight and take advantage of optimizations provided in the networking stack. For example, per-packet load balancing techniques [21] cannot be employed at the network edge because TSO does not work on a per-packet basis. TSO, commonly supported in OSes and NICs, allows for large TCP segments (typically 64 KB in size) to be passed down the networking stack to the NIC. The NIC breaks the segments into MTU-sized packets and copies and computes header data, such as sequence numbers and checksums. When TSO is disabled, a host incurs 100% CPU utilization and can only achieve around 5.5 Gbps [48]. Therefore, per-packet schemes are unlikely to scale to fast networks without hardware support. Limiting overhead by increasing the MTU is difficult because VMs, switches, and routers must all be configured appropriately, and traffic leaving the datacenter must use normal 1500 byte packets. Furthermore, per-packet schemes [21, 28] are likely to introduce significant reordering into the network.

Another possibility is to load balance on flowlets [9, 40]. A flow is comprised of a series of bursts, and a flowlet is created when the inter-arrival time between two packets in a flow exceeds a threshold inactivity timer. In practice, inactivity timer values are between 100-500 μs [9]. These values intend to strike a good balance between load balancing on a sub-flow level and acting as a

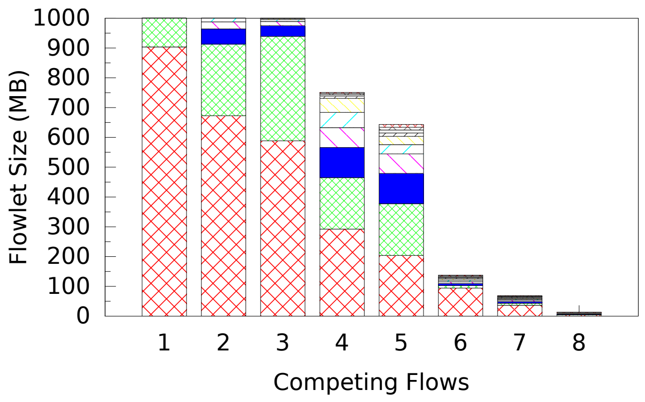


Figure 1: Stacked histogram of flowlet sizes (in MB) for a 1 GB `scp` file transfer. We vary the number of `nuttcp` background flows and denote them as *Competing Flows*. The size of each flowlet is shown within each bar, and flowlets are created whenever there is a 500 µs delay between segments. The top 10 flowlet sizes are shown here. We also analyzed the results of a 1 GB `nuttcp`, `ftp`, and a simple custom client/server transfer and found them to be similar.

buffer to limit reordering between flowlets. Flowlets are derived from traffic patterns at the sender, and in practice this means the distribution of flowlet sizes is not uniform. To analyze flowlet sizes, a simple experiment is shown in Figure 1. We connect a sender and a receiver to a single switch and start an `scp` transfer designed to emulate an elephant flow. Meanwhile, other senders are hooked up to the same switch and send to the same receiver. We vary the number of these competing flows and show a stacked histogram of the top 10 flowlet sizes for a 1 GB `scp` transfer with a 500 µs inactivity timer. The graph shows flowlet sizes can be quite large, with more than half the transfer being attributed to a single flowlet for up to 3 competing flows. Using a smaller inactivity timer, such 100µs, helps (90% of flowlet sizes are 114KB or less), but does not prevent a long tail: 0.1% of flowlets are larger than 1 MB, with the largest ranging from 2.1-20.5 MB. Collisions on large flowlet sizes can lead to congestion. The second problem with flowlets is that small inactivity thresholds, such as 100 µs, can lead to significant reordering. Not only does this impact TCP performance (profiled in §3.5), but it also needlessly breaks small flows into several flowlets. With only one flow in the network, we found a 50 KB mice flow was broken into 4-5 flowlets on average. Small flows typically do not need to be load balanced on a sub-flow level and need not be exposed to reordering.

The shortcomings of the previous approaches lead us to reconsider on what granularity load balancing should occur. Ideally, sub-flow load balancing should be done on near uniform sizes. Also, the unit of load balancing should be small to allow for fine-grained load balancing, but not so small as to break small flows into many pieces or as to be a significant computational burden. As a result, we propose load balancing on 64 KB units of data we call *flowcells*. Flowcells have a number of advantages. First, the maximum segment size supported by TSO is 64 KB, so flowcells provide a natural interface to high speed optimizations provided by the NIC and OS and can scale to fast networking speeds. Second, an overwhelming fraction of mice flows are less than 64 KB in size and thus do not have to worry about reordering [16, 37, 47]. Last, since most bytes in datacenter networks originate from elephant flows [10, 16, 47], this ensures that a significant portion of datacenter traffic is routed on uniform sizes. While promising, this approach must combat reordering to be effective. Essentially we make a trade-off: we provide line rate load balancing in the most effective manner as to avoid congestion and then handle reordering head-on at the receiver.

End-to-End vs Per-Hop Multipathing The last design consideration is whether multipathing should be done on a local, per-hop level (e.g., ECMP), or on a global, end-to-end level. In Presto, we choose the latter: pre-configured end-to-end paths are allocated in the network and path selection (and thus multipathing) is realized by having the network edge place flowcells onto these paths. Presto can be used to load-balance in an ECMP style per-hop manner but the choice of end-to-end multipathing provides additional benefits due to greater control on how flowcells are mapped to paths. Per-hop multipathing can be inefficient under asymmetric topologies [89], and load-balancing on a global end-to-end level can allow for weighted scheduling at vSwitch to rebalance traffic. This is especially important when failure occurs. The second benefit is that flowcells can be assigned over multiple paths very evenly by iterating over paths in a round-robin, rather than randomized, fashion.

3.2.2 Reordering Challenges

Due to the impact of fine-grained, flowcell-based load balancing, Presto must account for reordering. Here, we highlight reordering challenges. The next section shows how Presto deals with these concerns.

Reordering's Impact on TCP The impact of reordering on TCP is well-studied [54, 69]. Duplicate acknowledgments caused by reordering can cause TCP to move to a more conservative sender state and reduce the sender's congestion window. Relying on parameter tuning, such as adjusting the DUP-ACK threshold, is not ideal because increasing the DUP-ACK threshold increases the time to recover from real loss. Other TCP settings such as Forward Acknowledgement (FACK) assume un-acked bytes in the SACK are lost and degrade performance under reordering. A scheme that introduces reordering should not rely on careful configuration of TCP parameters because (i) it is hard to find a single set of parameters that work effectively over multiple scenarios and (ii) datacenter tenants should not be forced to constantly tune their networking stacks. Finally, many reordering-robust variants of TCP have been proposed [19, 20, 88], but as we will show, GRO becomes ineffective under reordering. Therefore, reordering should be handled below the transport layer.

Computational Bottleneck of Reordering Akin to TSO, Generic Receive Offload (GRO) mitigates the computational burden of receiving 1500 byte packets at 10 Gbps. GRO is implemented in the kernel of the hypervisor and its handler is called directly by the NIC driver. It is responsible for aggregating packets into larger segments that are pushed up to OVS and the TCP/IP stack. Because modern CPUs use aggressive prefetching, the cost of receiving TCP data is now dominated by per-packet, rather than per-byte, operations. As shown by Menon [60], the majority of this overhead comes from buffer management and other routines not related to protocol processing, and therefore significant computational overhead can be avoided by aggregating "raw" packets from the NIC into a single `sk_buff`.[2] Essentially, spending a few cycles to aggregate packets within GRO creates less segments for TCP and prevents having to use substantially more cycles at higher layers in the networking stack.

To better understand the problems reordering causes, a brief description of the TCP receive

[2]Refer to [60] for detailed study and explanation

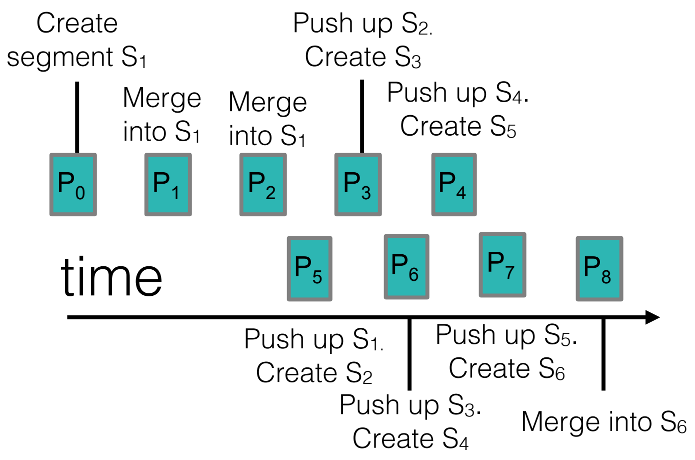


Figure 2: GRO pushes up small segments (S_i) during reordering.

chain in Linux follows. First, interrupt coalescing essentially allows the NIC to create an interrupt for a batch of packets [18, 63], which prompts the driver to poll the packets into an aggregation queue. Next, the driver invokes the GRO handler, located in the kernel, which *merges* the packets into larger segments. The merging continues, possibly across many polling events, until a segment reaches a threshold size, a certain age, or cannot be combined with the incoming packet. Then, the combined, larger segment is *pushed up* to the rest of the TCP/IP networking stack. The GRO process is done on a per-flow level. With GRO disabled, throughput drops to around 5.7-7.1 Gbps and CPU utilization spikes to 100% (§3.5 and [48]). Receive offload algorithms, whether in hardware (LRO) [38] or in software (GRO), are usually *stateless* to make them fast: no state is kept beyond the segment being merged.

We now uncover how GRO breaks down in the face of reordering. Figure 2 shows the impact of reordering on GRO. Reordering does not allow the segment to grow: each reordered packet cannot be merged with the existing segment, and thus the previously created segment must be pushed up. With extreme reordering, GRO is effectively disabled because small MTU-sized segments are constantly pushed up. This causes (i) severe computational overhead and (ii) TCP to be exposed to significant amounts of reordering. We term this the *small segment flooding* problem.

Determining where to combat the reordering problem has not previously taken the small segment flooding problem into account. Using a reordering buffer to deal with reordered packets is a common solution (e.g., TCP does this, other works re-sort in a shim layer below TCP [21]), but a buffer implemented above GRO cannot prevent small segment flooding. Implementing a buffer below GRO means that the NIC must be changed, which is (i) expensive and cumbersome to update and (ii) unlikely to help combat reordering over multiple interrupts.

In our system, the buffer is implemented in the GRO layer itself. We argue this is a natural location because GRO can directly control segment sizes while simultaneously limiting the impact of reordering. Furthermore, GRO can still be applied on packets pushed up from LRO, which means hardware doesn't have to be modified or made complex. Implementing a better GRO has multiple challenges. The algorithm should be fast and light-weight to scale to fast networking speeds. Furthermore, an ideal scheme should be able to distinguish loss from reordering. When a gap in sequence numbers is detected (e.g., when P_5 is received after P_2 in Figure 2), it is not obvious if this gap is caused from loss or reordering. If the gap is due to reordering, GRO should not push

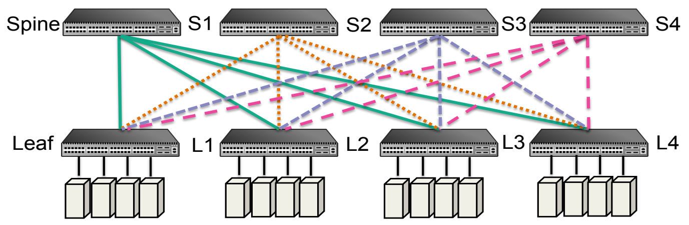


Figure 3: Our testbed: 2-tier Clos network with 16 hosts.

segments up in order to try to wait to receive the missing gap and merge the missing packets into a preestablished segment. If the gap is due to loss, however, then GRO should immediately push up the segments to allow TCP to react to the loss as fast as possible. Ideally, an updated GRO algorithm should ensure TCP does not perform any worse than a scheme with no reordering. Finally, the scheme should adapt to prevailing network conditions, traffic patterns and application demands.

3.3 Design

This section presents the design of Presto by detailing the sender, the receiver, and how the network adapts in the case of failures and asymmetry.

3.3.1 Sender

Global Load Balancing at the Network Edge In Presto, a centralized controller is employed to collect the network topology and disseminate corresponding load balancing information to the edge vSwitches. The goal of this design is to ensure the vSwitches, as a whole, can load balance the network in an even fashion, but without requiring an individual vSwitch to have detailed information about the network topology, updated traffic matrices or strict coordination amongst senders. At a high level, the controller partitions the network into a set of multiple spanning trees. Then, the controller assigns each vSwitch a unique forwarding label in each spanning tree. By having the vSwitches partition traffic over these spanning trees in a fine-grained manner, the network can load balance traffic in a near-optimal fashion.

The process of creating spanning trees is made simple by employing multi-stage Clos networks commonly found in datacenters. For example, in a 2-tier Clos network with ν spine switches, the controller can easily allocate ν disjoint spanning trees by having each spanning tree route through a unique spine switch. Figure 3 shows an example with four spine switches and four corresponding disjoint spanning trees. When there are γ links between each spine and leaf switch in a 2-tier Clos network, the controller can allocate γ spanning trees per spine switch. Note that 2-tier Clos networks are scalable because they easily support a large number of hosts with only a few high density spine switches [9]. In general, the controller ensures links in the network are equally covered by the allocated spanning trees. Once the spanning trees are created, the controller assigns a unique forwarding label for each vSwitch in every spanning tree and installs them into the network. Forwarding labels can be implemented in a variety of ways using technologies commonly deployed to forward on labels, such as MPLS [22], VXLAN [9, 50], or IP encapsulation [21]. In Presto, label switching is implemented with shadow MACs [2]. Shadow MACs implement label-switching for commodity Ethernet by using the destination MAC address as an opaque forwarding label that can easily be installed in L2 tables. Each vSwitch is assigned one shadow MAC per spanning tree. Note Shadow MACs are extremely scalable on existing chipsets because they utilize the large L2 forwarding table. For example, Trident II-based switches [27] have 288k L2 table entries and thus 8-way multipathing (i.e., each vSwitch has 8 disjoint spanning trees) can scale up to 36,000 physical servers.

Load Balancing at the Sender After the controller installs the shadow MAC forwarding rules into the network, it creates a mapping from each physical destination MAC address to a list of corresponding shadow MAC addresses. These mappings provide a way to send traffic to a specific destination over different spanning trees. The mappings are pushed from the controller to each vSwitch in the network, either on-demand or preemptively. In Presto, the vSwitch on the sender monitors outgoing traffic (i.e., maintains a per-flow counter in the datapath) and rewrites the destination MAC address with one of the corresponding shadow MAC addresses. The vSwitch assigns the same shadow MAC address to all consecutive segments until the 64 KB limit is reached. In order to load balance the network effectively, the vSwitch iterates through destination shadow MAC addresses in a round-robin fashion. This allows the edge vSwitch to load balance over the network in a very fine-grained fashion.

Sending each 64 KB worth of flowcells over a different path in the network can cause reordering and must be carefully addressed. To assist with reordering at the receiver (Presto's mechanisms for combatting reordering are detailed in the next section), the sender also includes a sequentially increasing *flowcell ID* into each segment. For example, in our setup the controller installs forwarding rules solely on the destination MAC address. ARP is handled in a centralized manner, so the source MAC address can be used to hold the flowcell ID. Other options are possible, e.g., some schemes include load balancing metadata in the reserved bits of the VXLAN header [39].[3] Note that since the flowcell ID and the shadow MAC address are modified before a segment is handed to the NIC, the TSO algorithm in the NIC replicates these values to all derived MTU-sized packets.

3.3.2 Receiver

The main challenge at the receiver is dealing with reordering that can occur when different flowcells are sent over different paths. The high level goal of our receiver implementation is to mitigate the effects of the small segment flooding problem by (i) not so aggressively pushing up segments if they cannot be merged with an incoming packet and (ii) ensuring that segments pushed up are delivered in order.

Mitigating Small Segment Flooding Let's use Figure 2 as a motivating example on how to combat the small segment flooding problem. Say a polling event has occurred, and the driver retrieves 9 packets from the NIC (P_0-P_8). The driver calls the GRO handler, which tries to merge consecutive packets into larger segments. The first three packets (P_0-P_2) are merged into a segment, call it S_1 (note: in practice S_1 already contains in order packets received before P_0). When P_5 arrives, a new segment S_2, containing P_5, should be created. Instead of pushing up S_1 (as is done currently), both segments should be kept. Then, when P_3 is received, it can be merged into S_1. Similarly, P_6 can be merged into S_2. This process can continue until P_4 is merged into S_1. At this point, the gap between the original out-of-order reception (P_2-P_5) has been filled, and S_1 can be pushed up and S_2

[3]In our implementation, TCP options hold the flowcell ID for simplicity and ease of debugging.

Algorithm 1 Pseudo-code of Presto GRO `flush` function

```
 1: for each flow f do
 2:     for S ∈ f.segment_list do
 3:         if f.lastFlowcell == getFlowcell(S) then
 4:             f.expSeq ← max(f.expSeq, S.endSeq)
 5:             pushUp(S)
 6:         else if getFlowcell(S) > f.lastFlowcell then
 7:             if f.expSeq == S.startSeq then
 8:                 f.lastFlowcell ← getFlowcell(S)
 9:                 f.expSeq ← S.endSeq
10:                 pushUp(S)
11:             else if timeout(S) then
12:                 f.lastFlowcell ← getFlowcell(S)
13:                 f.expSeq ← S.endSeq
14:                 pushUp(S)
15:             end if
16:         else
17:             pushUp(S)
18:         end if
19:     end for
20: end for
```

can continue to grow. This means the size of the segments being pushed up is increased, and TCP is not exposed to reordering.

The current default GRO algorithm works as follows. An interrupt by the NIC causes the driver to poll (multiple) packets from the NIC's ring buffer. The driver calls the GRO handler on the received batch of packets. GRO keeps a simple doubly linked list, called `gro_list`, that contains segments, with a flow having at most one segment in the list. When packets for a flow are received in-order, each packet can be merged into the flow's preexisting segment. When a packet cannot be merged, such as with reordering, the corresponding segment is pushed up (ejected from the linked list and pushed up the networking stack) and a new segment is created from the packet. This process is continued until all packets in the batch are serviced. At the end of the polling event, a `flush` function is called that pushes up all segments in the `gro_list`.

Our GRO algorithm makes the following high-level changes. First, multiple segments can be kept per flow, and each flow contains a doubly linked list of its segments (called `segment_list`). To ensure the merging process is fast each linked list is kept in a hash table (keyed on flow). When an incoming packet cannot be merged with any existing segment, the existing segments are kept and a new segment is created from the packet. New segments are added to the head of the linked list so that merging subsequent packets is typically $\mathcal{O}(1)$. When the merging is completed over all packets in the batch, the `flush` function is called. The `flush` function decides whether to push segments up or to keep them. Segments may be kept so reordered packets still in flight have enough time to arrive and can then be placed in order before being pushed up. Reordering can cause the linked lists to become slightly out-of-order, so at the beginning of `flush` an insertion sort is run to help easily decide if segments are in order.

The pseudo-code of our `flush` function is presented in Algorithm 1. For each flow, our algorithm keeps track of the next expected in-order sequence number (`f.expSeq`) and the corresponding flowcell ID of the most recently received in-order sequence number (`f.lastFlowcell`). The `flush` function iterates over the sorted segments (S), from lowest sequence number to highest sequence number, in the `segment_list` (line 2). The rest of the code is presented in the subsections that follow.

How to Differentiate Loss from Reordering? In the case of no loss or reordering, our algorithm keeps pushing up segments and updating state. Lines 3-5 deal with segments from the same flowcell ID, so we just need to update `f.expSeq` each time. Lines 6-10 represent the case when the current flowcell ID is fully received and we start to receive the next flowcell ID. The problem, however, is when there is a gap that appears between the sequence numbers of the segments. When a gap is encountered, it isn't clear if it is caused from reordering or from loss. If the gap is due to reordering, our algorithm should be conservative and try to wait to receive the packets that ”fill in the gap” before pushing segments up to TCP. If the gap is due to loss, however, then we should push up the segments immediately so that TCP can react to the loss as quickly as possible.

To solve this problem, we leverage the fact that all packets carrying the same flowcell ID traverse the same path and should be in order. This means incoming sequence numbers can be monitored to check for gaps. A sequence number gap within the same flowcell ID is assumed to be a loss, and not reordering, so those packets are pushed up immediately (lines 3-5). Note that because a flowcell consists of many packets (a 64KB flowcell consists of roughly 42 1500 byte packets), when there is a loss, it is likely that it occurs within flowcell boundaries. The corner case, when a gap occurs on the flowcell boundary, leads us to the next design question.

How to Handle Gaps at Flowcell Boundaries? When a gap is detected in sequence numbers at flowcell boundaries, it is not clear if the gap is due to loss or reordering. Therefore, the segment should be held long enough to handle reasonable amounts of reordering, but not so long that TCP cannot respond to loss promptly. Previous approaches that deal with reordering typically employ a large static timeout (10ms) [21]. Setting the timeout artificially high can handle reordering, but hinders TCP when the gap is due to loss. Setting a lower timeout is difficult because it depends on many dynamic factors such as delays between segments at the sender, amount of network congestion in different paths, and traffic patterns (multiple flows received at the same NIC affect inter-arrival time). As a result, we devise an adaptive timeout scheme, which monitors recent reordering events and sets a dynamic timeout value accordingly. Presto tracks cases when there is reordering, but no loss, on flowcell boundaries and keeps an exponentially-weighted moving average (EWMA) over these times. Presto then applies a timeout of $\alpha * \text{EWMA}$ to a segment when a gap is detected on flowcell boundaries. Here α is an empirical parameter that allows for timeouts to grow. As a further optimization, if a segment has timed out, but a packet has been merged into that segment in the last $\frac{1}{\beta} * \text{EWMA}$ time interval, then the segment is still held in hopes of preventing reordering. We find α and β work over a wide range of parameters and set both of them to 2 in our experiments. A timeout firing is dealt with in lines 11-15.

3.3.3 Failure Handling and Asymmetry

When failures occur, Presto relies on the controller to update the forwarding behavior of the affected vSwitches. The controller can simply prune the spanning trees that are affected by the failure, or more generally enforce a weighted scheduling algorithm over the spanning trees. Weighting allows for Presto to evenly distribute traffic over an asymmetric topology. Path weights can be implemented in a simple fashion by duplicating shadow MACs used in the vSwitch's round robin scheduling algorithm. For example, assume we have three paths in total (p_1, p_2 and p_3) and their updated weights are 0.25, 0.5 and 0.25 respectively. Then the controller can send the sequence of p_1, p_2, p_3, p_2 to the vSwitch, which can then schedule traffic over this sequence in a round robin fashion to realize the new path weights. This way of approximating path weights in the face of network asymmetry is similar to WCMP [89], but instead of having to change switch firmware and use scarce on-chip SRAM/TCAM entries, we can push the weighted load balancing entirely to the network edge.

As an added optimization, Presto can leverage any fast failover features that the network supports, such as BGP fast external failover, MPLS fast reroute, or OpenFlow failover groups. Fast failover detects port failure and can move corresponding traffic to a predetermined backup port. [4] This ensures traffic is moved away from the failure rapidly and the network remains connected when redundant links are available. Moving to backup links causes imbalance in the network, so Presto relies on the controller learning of the network change, computing weighted multipath schedules, and disseminating the schedules to the vSwitches.

3.4 Methodology

Implementation We implemented Presto in Open vSwitch v2.1.2 and Linux kernel v3.11.0. In OVS, we modified 5 files and ∼600 lines of code. For GRO, we modified 11 files and ∼900 lines of code.

Testbed We conducted our experiments on a physical testbed consisting of 16 IBM System x3620 M3 servers with 6-core Intel Xeon 2.53GHz CPUs, 60GB memory, and Mellanox ConnectX-2 EN 10GbE NICs. The servers were connected in a 2-tier Clos network topology with 10Gbps IBM RackSwitch G8264 switches, as shown in Figure 3.

Experiment Settings We ran the default TCP implementation in the Linux kernel (TCP CUBIC) and set `tcp_sack`, `tcp_fack`, `tcp_low_latency` to 1 unless otherwise noted. Further, we tuned the host RSS and IRQ affinity settings and kept them the same in all experiments.

Workloads We evaluate Presto with a set of synthetic and realistic workloads. Similar to previous works [6, 77], our synthetic workloads include:

Shuffle: Each server in the testbed sends 1GB data to every other server in the testbed in random order. Each host sends two flows at a time. This workload emulates the shuffle behavior of MapReduce/Hadoop workloads.

Stride(8): We index the servers in the testbed from left to right. In stride(8) workload, server[i] sends to server[(i+8) mod 16].

Random: Each server sends to a random destination not in the same pod as itself. Multiple senders can send to the same receiver.

Random Bijection: Each server sends to a random destination not in the same pod as itself. Different from random, each server only receives data from one sender.

Finally, we also evaluate Presto with trace-driven workloads from real datacenter traffic [47].

[4] Hardware failover latency ranges from several to tens of milliseconds

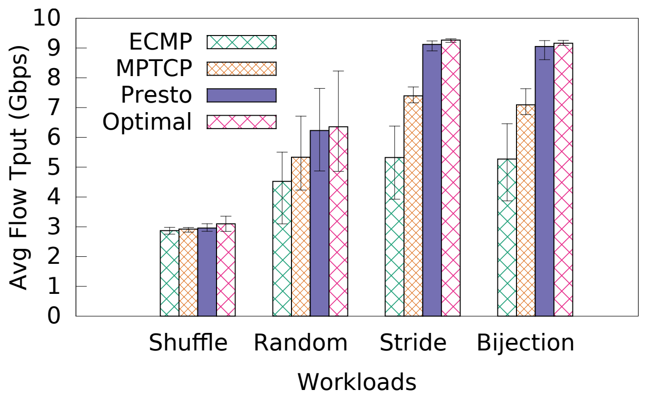


Figure 4: Elephant flow's throughputs of ECMP, MPTCP, Presto and Optimal in Shuffle, Random, Stride and Random Bijection workloads.

Performance Evaluation We compare Presto to ECMP, MPTCP, and a single non-blocking switch used to represent an optimal scenario. ECMP is implemented by enumerating all possible end-to-end paths and randomly selecting a path for each flow. MPTCP uses ECMP to determine the paths of each of its sub-flows. The MPTCP implementation is still under active development, and we spent significant effort in finding the most stable configuration of MPTCP on our testbed. Ultimately, we found that Mellanox `mlx_en4` driver version 2.2, MPTCP version 0.88, subflow count set to 8, OLIA congestion control algorithm, and configured buffer sizes as recommended by [76] gave us the best trade-offs in terms of throughput, latency, loss and stability. Unfortunately, despite our efforts, we still occasionally witness some stability issues with MPTCP that we believe are due to implementation bugs.

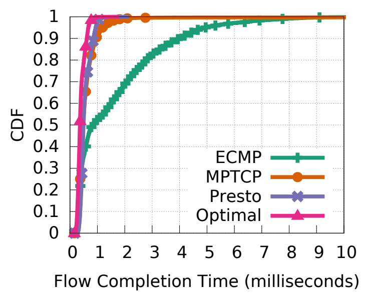
We evaluate Presto on various performance metrics, including: throughput (measured by `nuttcp`), round trip time (a single TCP packet, measured by `sockperf`), mice flow completion time (time to send a 50 KB flow and receive an application-layer acknowledgement), packet loss (measured from switch counters), and fairness (Jain's fairness index over flow throughputs). Mice flows are sent every 100 ms and elephant flows last 10 seconds. Each experiment is run for 10 seconds over 20 runs. Error bars on graphs denote the highest and lowest value over all runs.

3.5 Evaluation

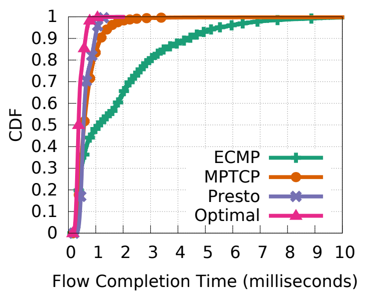
In this section, we analyze the performance of Presto for (i) synthetic workloads, (ii) trace-driven workloads, (iii) workloads containing north-south cross traffic, and (iv) failures. All tests are run on the topology in Figure 3.

Synthetic Workloads Figure 4 shows the average throughputs of elephant flows in the shuffle, random, stride and random bijection workloads. Presto's throughput is within 1-4% of Optimal over all workloads. For the shuffle workload, ECMP, MPTCP, Presto and Optimal show similar results because the throughput is mainly bottlenecked at the receiver. In the non-shuffle workloads, Presto improves upon ECMP by 38-72% and improves upon MPTCP by 17-28%.

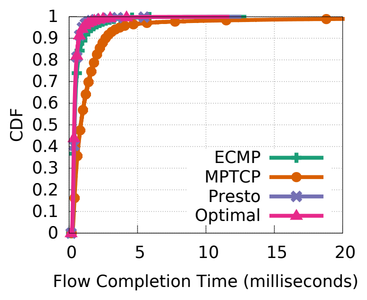
Figure 5 shows the mice flow completion time (FCT) in the workloads. The stride and random bijection workloads are non-blocking, and hence the latency of Presto closely tracks Optimal:



(a) Stride



(b) Random Bijection



(c) Shuffle

Figure 5: Mice FCT of ECMP, MPTCP, Presto and Optimal in stride, random bijection, and shuffle workloads.

Percentile	ECMP	Optimal	Presto
50%	1.0	−12%	−9%
90%	1.0	−34%	−32%
99%	1.0	−63%	−56%
99.9%	1.0	−61%	−60%

Table 1: Mice (<100KB) FCT in trace-driven workload [47]. Negative numbers imply shorter FCT.

the 99.9th percentile FCT for Presto is within 350 μs for these workloads. MPTCP and ECMP suffer from congestion, and therefore the tail FCT is much worse than Presto: ECMP's 99.9th percentile FCT is over 7.5x worse ($\sim$11ms) and MPTCP experiences timeout (because of higher loss rates and the fact that small sub-flow window sizes from small flows can increase the chances of timeout [76]).[5] The difference in the random and shuffle workloads is less pronounced (we omit random due to space constraints). In these workloads elephant flows can collide on the last hop output port, and therefore mice FCT is mainly determined by queuing latency. In shuffle, the 99.9th percentile FCT for ECMP, Presto and Optimal are all within 10% (MPTCP again experiences TCP timeout) and in random, the 99.9th percentile FCT of Presto is within 25% of Optimal while ECMP's is 32% worse than Presto.

Trace-driven Workload We evaluate Presto using a trace-driven workload based on traffic patterns measured in [47]. Each server establishes a long-lived TCP connection with every other server in the testbed. Then each server continuously samples flow sizes and inter-arrival times and each time sends to a random receiver that is not in the same rack. We scale the flow size distribution by a factor of 10 to emulate a heavier workload. Mice flows are defined as flows that are less than 100 KB in size, and elephant flows are defined as flows that are greater than 1 MB. The mice FCT, normalized to ECMP, is shown in Table 1. Compared with ECMP, Presto has similar performance at the 50th percentile but reduces the 99th and 99.9th percentile FCT by 56% and 60%, respectively. Note MPTCP is omitted because its performance was quite unstable in workloads featuring a large number of small flows. The average throughput (not shown) for Presto tracks Optimal (within 2%), and improves upon ECMP by over 10%.

[5] We used the Linux default (200ms) and trimmed graphs for clarity

Percentile	ECMP	Optimal	Presto	MPTCP
50%	1.0	−34%	−20%	−12%
90%	1.0	−83%	−79%	−73%
99%	1.0	−89%	−86%	−73%
99.9%	1.0	−91%	−87%	TIMEOUT

Table 2: FCT comparison (normalized to ECMP) with ECMP load balanced north-south traffic.

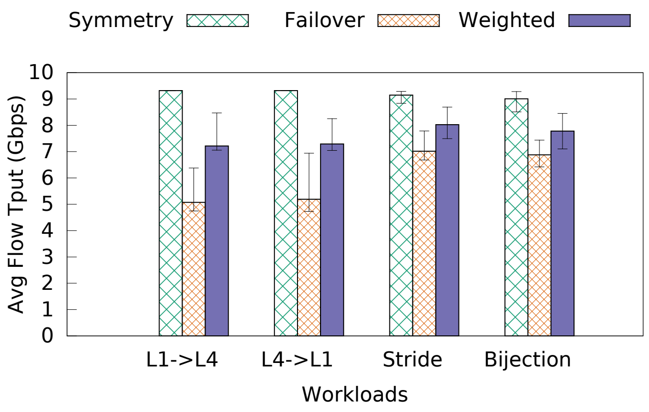


Figure 6: Presto's throughput in symmetry, fast failover and weighted multipathing stages for different workloads.

Impact of North-South Cross Traffic Presto load balances on "east-west" traffic in the datacenter, i.e., traffic originating and ending at servers in the datacenter. In a real datacenter environment "north-south" traffic (i.e., traffic with an endpoint outside the datacenter) must also be considered. To study the impact of north-south traffic on Presto, we attach an additional server to each spine switch in our testbed to emulate remote users. The 16 servers establish a long-lived TCP connection with each remote user. Next, each server starts a flow to a random remote user every 1 millisecond. This emulates the behavior of using ECMP to load balance north-south traffic. The flow sizes for north-south traffic are based on the distribution measurement in [41]. The throughput to remote users is limited to 100Mbps to emulate the limitation of an Internet WAN. Along with the north-south flows, a stride workload is started to emulate the east-west traffic. The east-west mice FCT is shown in Table 2 (normalized to ECMP). ECMP, MPTCP, Presto, and Optimal's average throughput is 5.7, 7.4, 8.2, and 8.9Gbps respectively. The experiment shows Presto can gracefully co-exist with north-south cross traffic in the datacenter.

Impact of Link Failure Finally, we study the impact of link failure. Figure 6 compares the throughputs of Presto when the link between spine switch S1 and leaf switch L1 goes down. Three stages are defined: symmetry (the link is up), failover (hardware fast-failover moves traffic from S1 to S2), and weighted (the controller learns of the failure and prunes the tree with the bad link). Despite the asymmetry in the topology, Presto still achieves reasonable average throughput at each stage. Figure 7 shows the round trip time of each stage in a random bijection workload. Workload L1→L4 is when each node connected to L1 sends to one node in L4 (L4→L1 is the opposite). Due to the fact that the network is no longer non-blocking after the link failure, failover and weighted

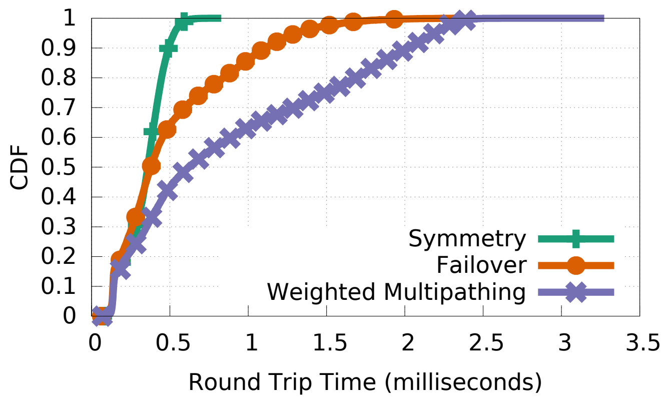


Figure 7: Presto's RTT in symmetry, fast failover and weighted multipathing stages in random bijection workload.

multipathing stages have larger round trip time.

4 Understanding Modern Web Service Deployment in EC2 and Azure

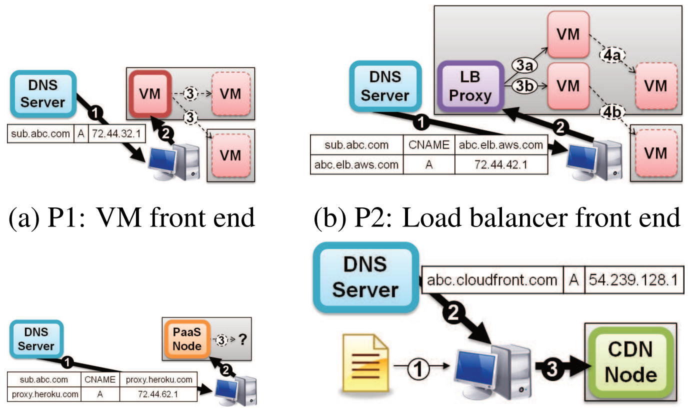
4.1 Introduction

An increasingly large fraction of Internet services are hosted on a cloud computing system such as Amazon EC2 or Windows Azure. But to date, no in-depth studies about cloud usage by Internet services has been performed. We provide a detailed measurement study to shed light on how modern web service deployments use the cloud and to identify ways in which cloud-using services might improve these deployments. Our results show that: 4% of the Alexa top million use EC2/Azure; there exist several common deployment patterns for cloud-using web service front ends; and services can significantly improve their wide-area performance and failure tolerance by making better use of existing regional diversity in EC2. Driving these analyses are several new datasets, including one with over 34 million DNS records for Alexa websites and a packet capture from a large university network.

4.2 Measurement Scope & Datasets

Public IaaS clouds, such as Amazon EC2, Windows Azure, and Rackspace, allow tenants to dynamically rent virtual machine (VM) instances with varying CPU, network, and storage capacity. Cloud tenants have the option of renting VMs in one or more geographically distinct data centers, or *regions*. Some clouds, such as EC2, further divide these regions into multiple distinct availability *zones*. Each zone has separate compute and power infrastructure to make certain failure modes zone-specific and to allow cloud tenants to replicate their deployments across multiple zones for smooth fail-over.

Beyond simple VMs, IaaS providers, as well as third parties, offer a wide-range of value-added features: load balancers (e.g., Amazon Elastic Load Balancer and Azure Traffic Manager), platform-


DNS Server
VM
VM
VM
sub.abc.com | A | 72.44.32.1
LB Proxy
3a
3b
4a
4b
sub.abc.com | CNAME | abc.elb.aws.com
abc.elb.aws.com | A | 72.44.42.1
PaaS Node
sub.abc.com | CNAME | proxy.heroku.com
proxy.heroku.com | A | 72.44.62.1
abc.cloudfront.com | A | 54.239.128.1
CDN Node

(a) P1: VM front end (b) P2: Load balancer front end

(c) P3: PaaS front end (d) P4: Leverage CDN

Figure 8: Deployment patterns for web services.

as-a-service environments (e.g., Amazon Elastic Beanstalk, Heroku, and Azure Cloud Services), content-distribution networks (e.g., Amazon CloudFront and Azure Media Services), DNS hosting (e.g., Amazon route53), etc. The result is a complex ecosystem of interdependent systems operating at multiple layers of abstraction, and, in turn, a large variety of possible deployment patterns for cloud tenants. In this paper, we study four popular deployment patterns. We describe these using a series of examples.

In Figure 8, we show the steps involved in a client accessing an EC2-hosted web service that is using one or more of the aforementioned features. When a client wants to access a web service, it first performs a DNS lookup of the service's domain name. The response may contain an IP address associated with a VM (deployment pattern *P1*), a load balancer (*P2*), or a platform-as-a-service (PaaS) node (*P3*). With *P2*, the client request is subsequently directed to a VM[6]. Tenants using *P1–P3* may also rely on additional VMs or systems (dashed lines) to handle a client's request; these additional components may or may not be in the same region or availability zone (indicated by the gray boxes). An object returned to a client (e.g., a web page) may sometimes require the client to obtain additional objects (e.g., a video) from a content-distribution network (*P4*).

We focus on studying the front end portions of web service deployments within the above four deployment patterns (indicated by the thicker lines in Figure 8). These portions are encountered within the initial few steps of a client making a request. We leave an exploration of deployment/usage patterns covering the later steps (e.g. back-end processing) for future work.

4.2.1 Datasets

We use two primary datasets: (*i*) a list of cloud-using subdomains derived from Alexa's list of the top 1 million websites, and (*ii*) packet traces captured at the border of the UW-Madison campus network. Both datasets leverage the fact that EC2 [30] and Azure [15] publish a list of the public IPv4 address ranges associated with their IaaS cloud offerings. Below, we provide details on our Alexa subdomains and packet capture datasets. We augment these data sets with additional traces and active measurements to aid specific analyses; we describe these at the appropriate places in subsequent sections.

[6]Or PaaS nodes, as is done by Amazon Elastic Beanstalk and Azure Traffic Manager.

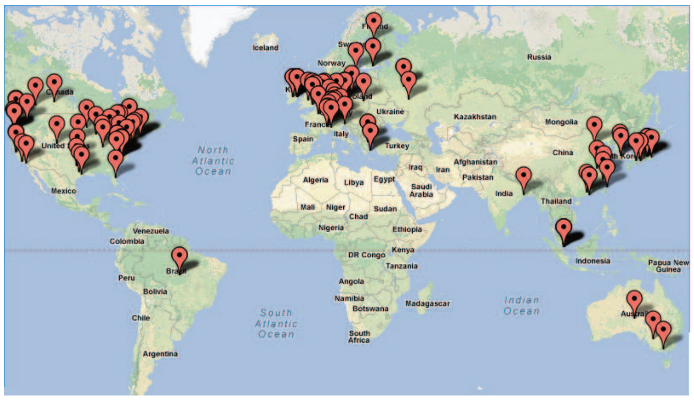


Figure 9: PlanetLab nodes used for DNS lookups

Top Cloud-Using Subdomains Dataset Our first dataset is a list of subdomains which use EC2 or Azure and are associated with domains on Alexa's list of the top 1 million websites [7]. We consider a subdomain to use EC2 or Azure if a DNS record for that subdomain contains an IP address that falls within EC2 or Azure's public IP address ranges.

To construct this dataset, we first identified the subdomains associated with each domain on Alexa's list of the top 1 million websites. We started with Alexa's top 1 million list from Feburary 6, 2013 and attempted to issue a DNS zone transfer (i.e., a DNS query of type AXFR) for each domain on the list. The query was successful for only about 80K of the domains. For the remaining domains, we used dnsmap [67] to identify subdomains by brute-force. Dnsmap uses a pre-defined word list, which we augmented with the word list from knock [49], to construct potential subdomain names. Dnsmap then runs DNS queries to check if the potential subdomains actually exist. This brute-force approach misses some subdomains, but it allows us to provide a lower bound on the number of subdomains which use public IaaS clouds and explore the deployment patterns of these known cloud-using subdomains. We distributed this task to 150 globally-distributed PlanetLab nodes, producing a list of 34 million valid subdomains.

To limit the list of subdomains to cloud-using subdomains, we performed a series of DNS lookups using the UNIX dig utility. We first performed a single DNS lookup from one PlanetLab node (chosen from our set of 150 nodes) for each subdomain. If the DNS record contained an IP address within EC2 or Azure's public IP ranges[7], we included it on our list of the top cloud-using subdomains. This resulted in a list of 713K cloud-using subdomains. We then performed a DNS lookup for each of the cloud-using subdomains on every node in a set of 200 globally-distributed PlanetLab nodes. Figure 9 shows the geographic location of these PlanetLab nodes, which are spread across North America, South America, Europe, Asia, and Australia. The queries were performed March 27-29, 2013. These distributed DNS queries help ensure that we gather a comprehensive set of DNS records for each cloud-using subdomain and capture any geo-location-specific cloud usage.

We refer to the list of cloud-using subdomains, and their associated DNS records, as the Alexa subdomains dataset.

Packet Capture Dataset Our second primary dataset is a series of packet traces captured at the border of the University of Wisconsin-Madison campus network[8]. We captured full IP packets

[7]We assume the IP address ranges published by EC2 and Azure are relatively complete.

[8]The university has seven /24 IP blocks and one /16 IP block

Cloud	Bytes	Flows
EC2	81.73	80.70
Azure	18.27	19.30
Total	100	100

Table 3: Percent of traffic volume and percent of flows associated with each cloud in the packet capture.

	EC2		Azure		Overall	
Protocol	Bytes	Flows	Bytes	Flows	Bytes	Flows
ICMP	0.01	0.03	0.01	0.18	0.01	0.06
HTTP (TCP)	16.26	70.45	59.97	65.41	24.24	69.48
HTTPS (TCP)	80.90	6.52	37.20	6.92	72.94	6.60
DNS (UDP)	0.11	10.33	0.10	11.59	0.11	10.58
Other (TCP)	2.40	0.40	2.41	1.10	2.40	0.60
Other (UDP)	0.28	0.19	0.31	14.77	0.28	3.00
Total	100	100	100	100	100	100

Table 4: Percent of traffic volume and percent of flows associated with each protocol in the packet capture.

whose source or destination IP address fell within the public address ranges published by EC2 and Azure. The capture was performed from Tuesday, June 26 to Monday, July 2, 2012 giving us a full week of traffic and a total of 1.4TB of data. The total Internet traffic averaged approximately 7Gbps during the capture, with about 1% of the traffic going to/coming from EC2 or Azure. Due to the relatively low rate of traffic being captured, no loss occurred during the capture process (according to tcpdump and counters reported by the border router). To protect user privacy, we anonymized the IP addresses of clients within the university network, and we only report aggregate statistics.

Since our traces contain full packets, we were able to perform an in-depth analysis of network and transport layer information (e.g., IP addresses, protocols, ports), application layer information (e.g., HTTP hostnames, HTTP content-type, HTTPS certificates), and packet payloads. We extracted relevant information from the traces using Bro [70], a network monitoring and traffic analysis tool. We refer to these traces as the packet capture dataset.

4.3 Web-Facing Cloud Tenants

In this section, we explore what applications are being hosted on public IaaS clouds. We start by analyzing the packet capture to identify the types of applications being hosted. This analysis suggests (unsurprisingly) that web applications represent a large, important set of cloud tenants. We then turn to examining which of the most popular websites are using clouds. We view popularity both globally, via the Alexa top website rankings, and locally, via the volume of traffic associated with each domain in the packet capture.

4.3.1 Protocols and Services

We first examine the fraction of bytes and flows in the packet capture that are associated with each cloud (Table 3). We only consider flows that were initiated within the university and destined for EC2 or Azure. We observe that the majority of cloud traffic, both as measured by volume and number of flows, is EC2-related: 81.73% of bytes (80.70% of flows) are associated with EC2, while Azure accounts for 18.27% of bytes (19.30% of flows).

Next, we use the packet capture to study the application-layer protocols used by cloud tenants. Table 4 shows the percentage of bytes (and flows) using a specific protocol relative to the total number of bytes (and flows) for EC2, Azure, and the capture as a whole.

We observe that more than 99% of bytes in the packet capture are sent and received using TCP, with less than 1% of bytes associated with UDP or ICMP. The vast majority of this TCP traffic is HTTP and HTTPS. The proportion of HTTPS traffic is far higher than that seen for general web services in the past (roughly 6% [1]); as we will show later, HTTPS traffic is dominated by cloud storage services. Interestingly, the majority of Azure's TCP traffic is HTTP (59.97%) while the majority of EC2's TCP traffic is HTTPS (80.90%)

The breakdown by flow count is less skewed towards TCP, with UDP flows accounting for 14% of flows in the packet capture. This is largely due to DNS queries, which account for 11% of flows but carry few bytes.

As one would expect, public IaaS clouds are also used for non-web-based services. In the packet capture, we find a small fraction of non-HTTP(S) TCP traffic and non-DNS UDP traffic going to both EC2 and Azure. This traffic includes SMTP, FTP, IPv6-in-IPv4, SSH, IRC, and other traffic that Bro could not classify.

Summary and implications While we analyze a single vantage point, our measurements suggest that web services using HTTP(S) represent an important set of WAN-intensive cloud tenants. The extent to which compute-intensive workloads (that may not result in a large impact on network traffic) are prevalent as cloud tenants remains an interesting open question. In the following sections we dig into what tenants are hosting web services on public clouds as well as diving deeper into their traffic patterns.

4.3.2 Popular Cloud-Using (Sub)Domains

Cloud-using Alexa domains We now consider what subset of the Alexa top 1 million websites use the cloud to (partly) host their services. Recall that Alexa provides an estimate of the most popular domains worldwide. Their ranking is based on the number of unique visitors and the number of page views over the last 3 months, aggregated at the domain level[9] [7]. Using our Alexa subdomains dataset, we can determine which Alexa sites are hosted in EC2/Azure.

We find that 40,333 (>4%) of the domains on Alexa's top 1 million list have a subdomain that uses EC2 and/or Azure. Under these domains, there are a total of 713,910 cloud-using subdomains. Note that these are lower bounds on cloud use, since our analysis approach (see §4.2.1) means we do not flag as cloud-using any domains that use a layer of indirection (e.g., via services like CloudFlare [26]) before requests are sent to EC2 or Azure.

[9]Except for domains hosting personal sites, e.g., wordpress.com, where subdomains are ranked individually.

Provider	# Domains (%)	# Subdomains (%)
EC2 only	3,277 (8.1%)	685,725 (96.1%)
EC2 + Other	34,721 (86.1%)	21,628 (3.0%)
Azure only	184 (0.5%)	6,328 (0.9%)
Azure + Other	1,858 (4.6%)	225 (<0.01%)
EC2 + Azure	293 (0.7%)	4 (<0.01%)
Total	40,333 (100.0%)	713,910 (100.0%)
EC2 total	38,291 (94.9%)	707,357 (99.1%)
Azure total	2,335 (5.8%)	6,557 (0.9%)

Table 5: Breakdown of domains and subdomains based on their use of EC2, Azure, and/or other hosting services.

Table 5 provides a breakdown of the domains and subdomains in terms of whether they use EC2, Azure, or other hosting services (the last indicating IP addresses not associated with EC2 or Azure). Note that "other" could in fact be public clouds besides EC2 and Azure. A subdomain is marked as *EC2 only* if it always resolves only to IP addresses within EC2; similarly for Azure. We mark a subdomain as *EC2+Azure*, *EC2+Other*, or *Azure+Other* if it resolves to IP addresses associated with the appropriate subset of EC2, Azure, and other. Domains are counted as *EC2 only* if all of their subdomains only use EC2; similarly for Azure. Domains are marked as *EC2+Azure*, *EC2+Other*, or *Azure+Other* if they have subdomains associated with the indicated subset of EC2, Azure, and other.

The vast majority of cloud-using domains (94.9%) use EC2, and the majority of these domains use other hosting for some of their subdomains (i.e., EC2 + Other). Only 5.8% of domains use Azure. Additionally, a small fraction (0.7%) of cloud-using domains use both EC2 and Azure; hence the *EC2 total* and *Azure total* rows in Table 5 sum to more than 100%. A list of the top 10 (by Alexa rank) EC2-using domains appears in Table 6. This list will be used in several later sections with results specific to EC2, which is why we excluded the four top Azure domains that would otherwise have been in the top 10: live.com, msn.com, bing.com, and microsoft.com.

The distribution of Alexa ranks for cloud-using domains is skewed: higher ranked domains are more likely to be cloud-using than lower ranked domains. Notably, 42.3% of cloud-using domains have ranks in the first 250,000 sites versus only 16.2% of the bottom 250K domains.

The most frequent prefix used by cloud-using subdomains in our Alexa subdomains dataset is www (3.3% of all cloud-using subdomains). The other top 10 prefixes (each $<1\%$) are, in order: m, ftp, cdn, mail, staging, blog, support, test, and dev. The majority of subdomains are hosted either only in the cloud or only elsewhere, although a small fraction (3%) appear to be hosted both on EC2 and other providers, what we might call a hybrid-cloud deployment.

High traffic volume domains We complement the above with an analysis of the top domains seen in the packet capture, as measured by traffic volume. We use Bro to extract hostnames within HTTP requests and common names within the server certificates embedded in HTTPS flows[10]. Aggregating the hostnames and common names by domain, we find 13,604 unique cloud-using domains: 12,720 use EC2 and 885 use Azure. Of these 13,604 domains, 6902 were also identified

[10]TLS encryption hides the hostname associated with the underlying HTTP requests, so we use the common names found in TLS server certificates as a proxy.

Rank	Domain	Total # Subdom	# EC2 Subdom
9	amazon.com	68	2
13	linkedin.com	142	3
29	163.com	181	4
35	pinterest.com	24	18
36	fc2.com	89	14
38	conduit.com	40	1
42	ask.com	97	1
47	apple.com	73	1
48	imdb.com	26	2
51	hao123.com	45	1

Table 6: Top 10 (by Alexa rank) EC2-using domains, their total number of subdomains, and the number of EC2-using subdomains.

EC2			Azure		
Domain	Rank	Traffic (%)	Domain	Rank	Traffic (%)
dropbox.com (d)	119	595.0 (68.21)	atdmt.com	11,128	27.0 (3.10)
netflix.com (d)	92	14.8 (1.70)	msn.com	18	20.9 (2.39)
truste.com (d)	15,458	9.2 (1.06)	microsoft.com	31	19.7 (2.26)
channel3000.com	29,394	6.4 (0.74)	msecnd.net	4,747	13.5 (1.55)
pinterest.com (d)	35	5.1 (0.59)	s-msn.com	25,363	12.5 (1.43)
adsafeprotected.com (d)	371,837	4.7 (0.53)	live.com	7	11.8 (1.35)
zynga.com	799	3.9 (0.44)	virtualearth.net	147,025	9.2 (1.06)
sharefile.com	20,533	3.6 (0.42)	dreamspark.com	35,223	7.1 (0.81)
zoolz.com	272,006	3.2 (0.36)	hotmail.com	2,346	6.3 (0.72)
echoenabled.com (d)	-	2.7 (0.31)	mesh.com	-	4.5 (0.52)
vimeo.com	137	2.3 (0.26)	wonderwall.com	-	3.2 (0.36)
foursquare.com	615	2.2 (0.25)	msads.net	-	2.5 (0.29)
sourcefire.com	359,387	1.9 (0.22)	aspnetcdn.com	111,859	2.3 (0.26)
instagram.com (d)	75	1.5 (0.17)	windowsphone.com	1,597	2.0 (0.23)
copperegg.com	122,779	1.5 (0.17)	windowsphone-int.com	-	2.0 (0.23)

Table 7: Domains with highest HTTP(S) traffic volumes (in GB) in the packet capture. Percentages are relative to the total HTTP(S) traffic across both clouds in the capture. Domains marked with (d) appeared on DeepField's Top 15 [52].

as cloud-using via the Alexa dataset; the remainder were not in the Alexa top 1 million. Table 7 lists the highest 15 such domains in terms of traffic volume. A few tenants are responsible for a large fraction of the traffic. Most notably, dropbox.com accounts for almost 70% of the combined HTTP(S) traffic volume. This also explains why HTTPS (used by dropbox.com) dominates HTTP in terms of volume (though not number of flows, refer to Table 4).

It is informative to compare our analysis of top EC2-using domains by traffic volume to the analysis by DeepField networks [52], which was conducted three months before we collected the packet capture. They used data from customers of their network analysis products. Seven of the domains on our top 15 list also appear within the top 15 on DeepField's list (indicated with a (d) in

Table 7).

Summary and implications A substantial fraction of the world's most popular websites rely in whole or in part on public IaaS clouds, especially EC2. Most cloud-using domains have some subdomains hosted on a cloud service while other subdomains are hosted elsewhere. Perhaps surprisingly a small, but noticeable fraction of subdomains use both a cloud and other hosting solutions. Finally, traffic volume appears to be dominated by a few cloud tenants (we discuss traffic patterns more next). Depending on how tenants' deploy their services (e.g., how many and which regions they use), these observations have implications for availability of web-based cloud-resident services. We explore the underlying deployments in more detail in §4.4.

4.4 Tenants' Deployment Posture

In this section, we attempt to understand *how* tenants use the cloud for deploying their front ends. We start by analyzing the deployment patterns employed by cloud-using (sub)domains. We first focus on the four patterns identified in Figure 8. We then quantify how many, and which, regions and availability zones are leveraged by cloud-using (sub)domains' front ends.

4.4.1 Deployment Patterns

In this section, we use the DNS records from our Alexa subdomains dataset and a variety of heuristics to detect and quantify usage of the deployment patterns outlined in Figure 8. Specifically, we estimate the use of virtual machines (VMs), platform-as-a-service (PaaS) environments, load balancers, content-distribution networks (CDNs), and domain name servers within the front ends of web services hosted in both EC2 and Azure. In general, we discuss EC2 and Azure separately because of differences in the cloud architectures.

VM front end in EC2 Each VM instance in an IaaS cloud combines a set of virtual resources (CPU core(s), memory, local storage, and network bandwidth) whose capacity depends on the instance type. In EC2 each instance is assigned an internal IP address within a region-specific private network; EC2 tenants may optionally assign a public (i.e., Internet-routable) IP address to a VM.

We identify usage of VMs as directly-reachable web service front ends—i.e., deployment pattern *P1* (Figure 8a)—by examining if the DNS query for an EC2-using subdomain directly returns an IP address (instead of a CNAME), which we then associate with a VM instance. We find that 505,578 (72%) EC2-using subdomains leverage front end VMs. Figure 10a shows a CDF of the number of front end VM instances used by each EC2-using subdomain; this CDF only includes subdomains which use front end VMs. We observe that about half of such subdomains use 2 front end VMs and 15% use 3 or more front end VMs.

Aggregating by domain, we find that 52% of EC2-using domains have at least one subdomain which uses at least one front end VM. If we sum the number of front end VMs used across all subdomains of a given domain, we find that 10% of domains which use front end VMs in EC2 use 3 or more front end VMs in total.

PaaS front end in EC2 PaaS systems offer a hosted environment for deploying web applications, avoiding the need for tenants to manage low-level system details. PaaS systems are frequently built atop existing IaaS infrastructure: e.g., Amazon's Elastic Beanstalk and Heroku both run

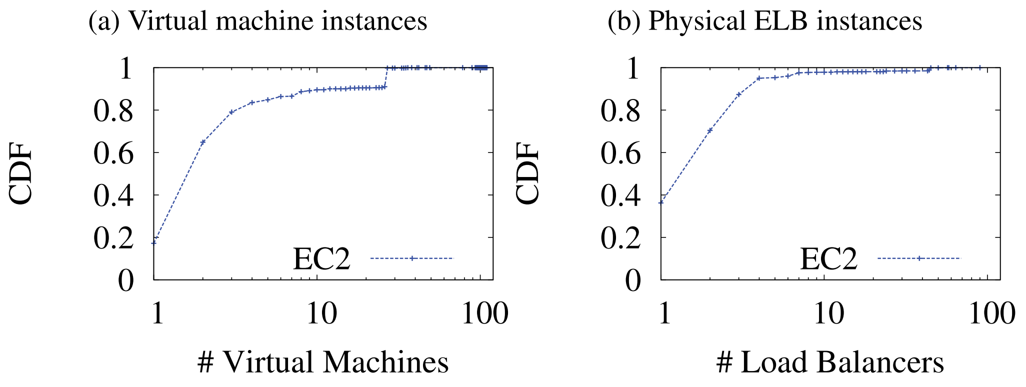


Figure 10: CDFs for the # of feature instances per subdomain (only includes subdomains which use the feature).

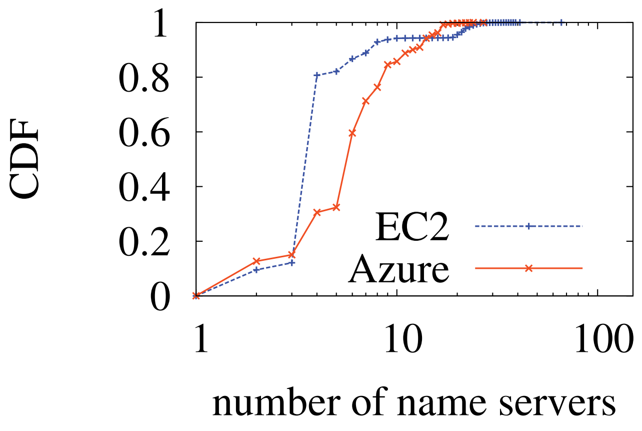


Figure 11: CDF of the # of DNS servers used per subdomain.

atop EC2. A Beanstalk environment always includes an Amazon Elastic Load Balancer (ELB) instance (discussed in more detail below), reflecting deployment pattern *P2* (Figure 8b, replace VMs with PaaS nodes). A Heroku environment may or may not include an ELB, reflecting usage of deployment patterns *P2* or *P3* (Figure 8), respectively.

We say that a subdomain uses Beanstalk or Heroku if the subdomain's DNS record has a CNAME that (*i*) includes 'elasticbeanstalk' or any of 'heroku.com', 'herokuapp', 'herokucom', and 'herokussl' and (*ii*) resolves to an IP in EC2's public IP address range. In the case of Heroku without ELB, the IPs to which the CNAME resolves represent PaaS nodes; we associate these IPs with the subdomain whose DNS record contains the corresponding CNAME.

A total of 201,666 (28%) EC2-using subdomains in our Alexa subdomains dataset contain a CNAME in their DNS record. Applying the above filters for PaaS, we find that 60,273 (8%) EC2-using subdomains use a front end PaaS environment in EC2. Of these, over 97% (59,991) are using Heroku; only 3% use Elastic Beanstalk. Amazon always includes an ELB in a Beanstalk environment, but Heroku only sometimes leverages ELBs—only 3% of subdomains (1,850) which use Heroku also use ELB. We therefore conclude that, in the case of EC2, PaaS systems are predominantly used according to deployment pattern *P3* (Figure 8c).

We now focus on the 58,141 (59,991 - 1,850) subdomains that use Heroku without ELB. We find that these are associated with just 94 unique IPs. Although we have no insight into the number of worker instances used by Heroku, this shows that Heroku is multiplexing PaaS functionality among a relatively large number of subdomains: in particular, we find that about one-third of subdomains using Heroku share the CNAME 'proxy.heroku.com'.

Load balancer front end in EC2 Load balancers divide traffic among a set of "worker" VMs or PaaS nodes, as reflected in deployment pattern *P2* (Figure 8b). Amazon Elastic Load Balancers (ELBs) are Amazon-managed HTTP proxies. An EC2 tenant requests an ELB in a specific region and subsequently associates VM instances, in one or more zones, with this ELB. The ELB automatically round-robins requests among zones and among the VMs in each zone. In fact, traffic is routed to zone-specific ELB proxies by rotating the order of ELB proxy IPs in DNS replies. ELB can also be used with PaaS, as discussed above.

When a subdomain uses an ELB, the subdomain's DNS record contains a CNAME ending in 'elb.amazonaws.com'; the CNAMEs resolve to IP addresses for one or more ELB proxies. We identify ELB-using subdomains in our Alexa subdomains dataset based on the presence of such CNAMEs; we refer to each distinct CNAME as a "logical ELB instance". We also associate with the subdomain the IPs of the specific ELB proxies to which the CNAME resolves; we refer to these as "physical ELB instances".

We find that 27,154 (4%) EC2-using subdomains use ELB as their front end. Of the subdomains that use ELB, 280 (1%) use it in the context of Elastic Beanstalk and 1,850 (6.8%) use it with Heroku. Aggregating by domain, we find that 9,851 (26%) EC2-using domains use front end ELB(s).

Across all ELB-using subdomains, we observe 15,703 physical ELB instances (i.e., distinct IPs associated with ELB CNAMEs). Hence, while each subdomain has its own logical ELB(s), the physical ELB proxies that perform the actual load balancing appear to be shared across multiple, even unrelated, subdomains. In particular, we analyzed the number of subdomains per physical ELB and found that ≈4% of the physical ELB instances are shared by 10 or more subdomains.

Figure 10b shows a CDF of the number of physical ELB instances associated with each subdomain; this CDF only includes subdomains which use ELB. We observe that about 95% of ELB-using subdomains are associated with 5 or fewer physical ELB instances. A few ELB-using subdomains (e.g., dl.outbrain.com and m.netflix.com) use many physical ELB instances: 58 and 90, respectively.

Content distribution networks We now focus on the use of CDNs, which we illustrated in deployment pattern *P4* (Figure 8d). Note that CDNs can be employed alongside any of the other three deployment patterns.

Both Microsoft and Amazon run their own CDNs, which we focus on studying. Amazon's CloudFront CDN uses a different public IP address range than the rest of EC2. Hence, we determine if a subdomain uses CloudFront by observing if its DNS records contain one or more IPs in CloudFront's IP range. Azure's CDN uses the same IP address ranges as other parts of Azure, so we detect whether a subdomain uses the Azure CDN based on whether a subdomain's DNS records contain CNAMEs with 'msecnd.net'.

We find 7,622 subdomains (corresponding to 5,988 domains) use CloudFront and 68 subdomains (corresponding to 54 domains) use Azure's CDN. Despite the much smaller number of domains using Azure's CDN, there is still a significant volume of traffic associated with msecnd.net in our packet capture dataset (Table 7). Azure's CDN is clearly being used within some Microsoft

Cloud	Feature	# Domains	# Subdomains	# Inst.
EC2	VM	19.9K (52.5%)	505.6K (71.5%)	28.3K
	ELB	9.9K (25.9%)	27.1K (3.8%)	15.7K
	BeanStalk (w/ ELB)	188 (0.5%)	280 (< 0.01%)	455
	Heroku (w/ ELB	622 (1.6%)	1.9K (0.3%)	2.4K
	Heroku (no ELB)	1.3K (3.5%)	58.1K (8.2%)	94
Azure	CS	863 (37.0%)	4.5K (68.3%)	790
	TM	52 (2.2%)	100 (1.5%)	78

Table 8: Summary of cloud feature usage.

properties, perhaps to host embedded content or cookies.

Domain name servers The first step in accessing a cloud-resident service is to resolve its name (Figure 8). In what follows, we examine cloud-resident subdomain's use of DNS, focusing on the extent to which they rely on cloud providers for DNS services as well.

We identifed the "location" of a cloud-using subdomain's authoritative name server(s) as follows: For each DNS record associated with a given subdomain in our Alexa subdomains dataset, we extract all the domains specified in the NS records. We then performed a DNS lookup on each of these domains from 50 globally-distributed PlanetLab nodes. We flushed and reset the cache of the local resolver between each DNS lookup, and we added the 'norecurse' flag to each DNS query to minimize the influence of caching. We compare the resulting IP addresses to the public IP address ranges for EC2, CloudFront, and Azure.

We observe a total of 23,111 name servers supporting the 713K cloud-using subdomains in our Alexa subdomains dataset. Many subdomains use the same name servers, leading to a smaller set of name servers than subdomains. Figure 11 shows a CDF for the number of name servers used by each cloud-using subdomain; we observe that nearly 80% of subdomains use 3 to 10 name servers. We categorize the name servers as follows: 2,062 were hosted in CloudFront, which appears to host Amazon's route53 DNS service as many of these name servers had 'route53' in their domain name; 1,239 were running inside EC2 VM instances; 22 were hosted inside Azure VM instances or Azure CS; and 19,788 were hosted outside any of EC2, CloudFront, or Azure;

The above analyses are summarized in Table 8 which shows how many (sub)domains in our Alexa subdomains dataset use each cloud feature. We also show the number of instances (identified by IP address) of that feature.

Analysis of top domains As notable exemplars, Table 9 gives a detailed breakdown of the cloud feature usage of the most popular (according to Alexa rankings) EC2-using domains. We observe that the majority of subdomains associated with the top domains have VM or ELB front ends. Of those using ELB front ends, amazon.com and fc2.com use ELB the most (i.e., there are more physical ELB IPs associated with these domains). Three of the top domains have subdomains which use a CDN, but only one of these domains uses the CloudFront CDN.

Summary and implications In summary, we find that the majority (71.5%) of EC2-using subdomains use a VM front end (deployment pattern *P1*); hence most EC2 tenants are using EC2 as a true IaaS cloud. Only a small fraction use an ELB front end (3.8%) or PaaS front end (8.5%). Due to limited use, failures of value-added features are unlikely to have a major impact on EC2-using subdomains. In Azure, we are able to identify the usage of VM, PaaS, or load balancing front ends

Rank	Domain	# Cloud Subdom	Front-end VM	Front-end PaaS	Front-end ELB	ELB IPs	Use CDN
9	amazon.com	2	0	1	2	27	0
13	linkedin.com	3	0	1	1	1	0
29	163.com	4	0	0	0	0	4*
35	pinterest.com	18	4	0	0	0	0
36	fc2.com	14	10	0	4	68	0
38	conduit.com	1	0	1	1	3	0
42	ask.com	1	1	0	0	0	0
47	apple.com	1	1	0	0	0	0
48	imdb.com	2	0	0	0	0	1
51	hao123.com	1	0	0	0	0	1*

Table 9: Cloud feature usage for the highest ranked EC2-using domains (* indicates use of a CDN other than CloudFront).

(we cannot distinguish which) for 70% of subdomains. A small fraction (1.5%) of Azure-using domains leverage TM to balance traffic across different front ends. The majority of DNS servers used by cloud-using subdomains reside outside of EC2 or Azure, giving subdomains the option of routing traffic to different resources (in another cloud or a private data center) in the event of cloud failure.

4.4.2 Region Usage

EC2 and Azure give tenants the choice of using one or more geographically distinct regions (i.e., data centers). Regions provide a mechanism for robustness in the case of catastrophic failures, e.g., regional power or service outages [12, 32, 33, 80]. In this section, we examine how many, and which, of the eight regions offered by each cloud provider are leveraged by the front ends of cloud-using (sub)domains.

We ascertain the region(s) used by each subdomain in the Alexa subdomains dataset by comparing the IP addresses associated with that subdomain against the per-region IP address ranges published by EC2 [30] and Azure [15]. We only consider IPs associated with VM, PaaS, and ELB/TM.

Figure 12a shows a CDF (note the Y-axis starts at 90%) of the number of regions used by each subdomain in the Alexa subdomains. Over 97% of EC2-using and 92% of Azure-using subdomains exclusively use one region. Across all domains (Figure 12b), the trend of low region usage is largely the same, although, the fraction of Azure-using domains that only use one region (83%) is smaller than the fraction of subdomains that only use one region (92%).

The number of (sub)domains (from the Alexa subdomains) in each region are shown in Table 10. We observe that the usage of EC2 regions is heavily skewed towards a few regions: 74% of EC2-using subdomains use *US East* and 16% use *Europe West*. Azure, relatively speaking, has a more even distribution of subdomains across regions, but each region has significantly fewer subdomains. The most used Azure regions are *US South* and *US North*.

Analysis of top domains We now focus on region usage of subdomains corresponding to the most popular (according to Alexa rankings) domains. Our analysis is summarized in Table 11. As with

Region	Location	# Dom	# Subdom
ec2.us-east-1	Virginia, USA	25,722	521,681
ec2.eu-west-1	Ireland	6,834	116,366
ec2.us-west-1	N. California, USA	3,950	40,548
ec2.us-west-2	Oregon, USA	1,548	15,635
ec2.ap-southeast-1	Singapore	1,800	20,871
ec2.ap-northeast-1	Tokyo, Japan	2,223	16,965
ec2.sa-east-1	São Paulo, Brazil	625	14,866
ec2.ap-southeast-2	Sydney, Australia	313	554
az.us-east	Virginia, USA	268	862
az.us-west	California, USA	161	558
az.us-north	Illinois, USA	590	2,071
az.us-south	Texas, USA	1,072	1,395
az.eu-west	Ireland	564	1,035
az.eu-north	Netherlands	573	1,205
az.ap-southeast	Singapore	379	632
az.ap-east	Hong Kong	333	502

Table 10: EC2 and Azure region usage Alexa subdomains

the rest of our results above, we see that in all but two cases, subdomains appear to use a single region. The exceptions are msn.com and microsoft.com, where 11 of the 89 subdomains and 4 of 11 subdomains, respectively, use two regions each. No popular subdomain uses three or more regions. We also note that in some cases, a domain may deploy different subdomains across different regions: e.g., live.com's 18 subdomains are spread across 3 regions. Contrarily, there are domains whose subdomains are all deployed in one region (e.g., pinterest.com).

Analysis of subdomain deployment vs. customer location An interesting question about cloud service deployment is whether subdomains are deployed near their customers? The answer to this question reveals whether current cloud services are deployed in an "optimal" manner, because deploying a service near customers usually leads to better client network performance (lower latency and higher throughput).

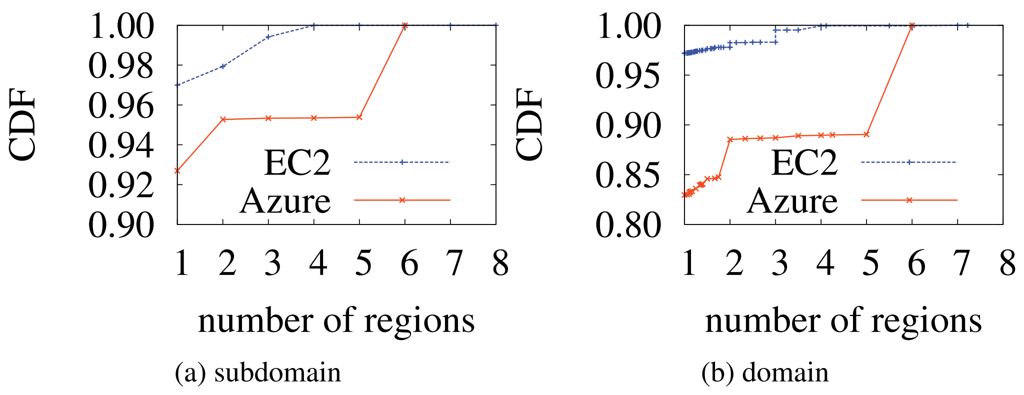


Figure 12: (a) CDF of the number of regions used by each subdomain (b) CDF of the average number of regions used by the subdomains of each domain.

Rank	Domain	# Cloud Subdom	Total # Regions	*k*=1	*k*=2
7	live.com	18	3	18	0
9	amazon.com	2	1	2	0
13	linkedin.com	3	2	3	0
18	msn.com	89	5	78	11
20	bing.com	1	1	1	0
29	163.com	4	1	4	0
31	microsoft.com	11	5	7	4
35	pinterest.com	18	1	18	0
36	fc2.com	14	2	14	0
42	ask.com	1	1	1	0
47	apple.com	1	1	1	0
48	imdb.com	2	1	2	0
51	hao123.com	1	1	1	0
59	go.com	4	1	4	0

Table 11: Region usage for the top cloud-using domains. The third column is the number of cloud-using subdomains; fourth is the total number of regions used by a domain; and the $k = 1$ and $k = 2$ columns are the number of subdomains which use one or two regions, respectively.

To answer this question, we leverage the client geo-location information provided by the Alexa web information service [8]. For example, at the time of writing, Alexa reported that 47% of clients accessing pinterest.com are from the United States, 10.4% from India, 3.2% from the United Kingdom, 3.1% from Canada, and 2.1% from Brazil. For each domain, we define the "customer country" as the country where the largest fraction of clients are located. We assume the customer country is the same for all of a website's subdomains, as Alexa does not track subdomains separately. For instance, the United States is the customer country for pinterest.com (and its subdomains) based on our definition.

We performed the analysis for all of the cloud-using subdomains (about 713K) in our dataset. Our measurement methodology was able to successfully identify approximately 538K (75% of the total) subdomains' customer country. We find that 252K (47%) subdomains' customer country is not the same as the country where this subdomain is hosted. Moreover, 174K (32%) subdomains are not even hosted on the same continent as the subdomains' customer country. This implies that a large fraction of web services are probably not deployed in an optimal manner in terms of network performance. We suspect that the current deployment posture is affected by computing, storage, and network costs and/or how long the cloud region has existed. In §4.5, we explore how much opportunity exists for improving wide-area performance through changes in region usage.

Summary and implications Our key finding in this section is that most popular domains and subdomains appear to be using a single region. This has significant implications on both the robustness and performance of cloud-using web services. From an availability perspective, an outage of EC2's *US East* region would take down critical components of at least 2.3% of the domains (61% of EC2-using domains) on Alexa's list of the top 1 million websites. This is a lower bound, as our results do not include dependencies between domains. From a performance perspective, our analysis of web service deployment and customer locations reveals that a considerable fraction of

Region	1st zone		2nd zone		3rd zone	
	#Dom	#Sub	#Dom	#Sub	#Dom	#Sub
ec2.us-east-1	16.1	419.0	6.2	155.4	9.5	292.9
ec2.us-west-1	1.6	33.2	3.0	37.4	N/A	N/A
ec2.us-west-2	0.9	13.4	1.0	9.6	0.8	7.3
ec2.eu-west-1	2.3	77.0	2.9	63.9	4.5	98.7
ec2.ap-northeast-1	0.4	3.7	1.3	11.3	1.5	12.9
ec2.ap-southeast-1	0.9	11.3	1.2	19.1	N/A	N/A
ec2.ap-southeast-2	0.2	0.3	0.2	0.3	N/A	N/A
ec2.sa-east-1	0.5	14.4	0.2	8.9	N/A	N/A

Table 12: Estimated number of domains and subdomains using various EC2 zones. Some regions only have 2 zones.

client traffic may travel farther than necessary due to suboptimal provisioning.

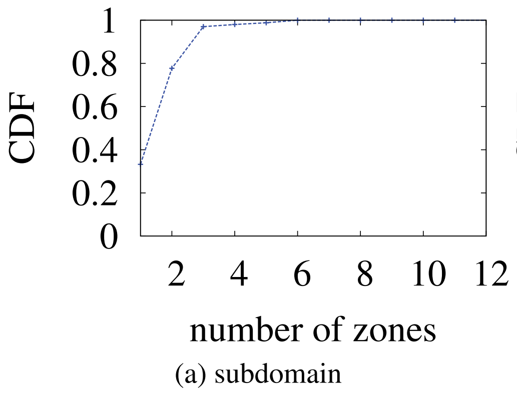
4.4.3 Availability Zone Usage

Within each region of EC2, cloud tenants have the choice of deploying across multiple zones. EC2 zones offer a means for improving service robustness as they are claimed to use separate compute, network, and power infrastructure so that a failure in any of these will not affect more than one zone. There seems to be no equivalent of zones in Azure.

We now focus on determining the zone deployment for EC2-using services' front ends. Unlike the regions, which are easily distinguished based on the IP address of a subdomain and the advertised ranges [15, 30], there is no direct way to associate an IP address to a zone. We therefore turn to cloud cartography techniques [79]. We use two methods to identify zones: network latency and proximity in the internal addresses to instances with a known zone (i.e., VMs we launched). Details about the two methods are presented in [41]. We combine the two zone identification methods to maximize the fraction of physical instances whose zone we can identify. We give preference to our address-proximity-based zone identifications, and use our latency-based identifications only for instances whose zone cannot be identified using the former method. Combining the two methods allows us to identify the EC2 availability zone for 87.0% of all physical EC2 instances in the Alexa subdomains dataset.

Table 12 summarizes the number of (sub)domains using each region and zone. In all but one region (Asia Pacific Southeast 2), we observe a skew in the number of subdomains using each zone in a region. Asia Pacific Northeast and US East regions have the highest skew across their three zones: 71% and 63% fewer subdomains, respectively, use the least popular zone in those regions compared to the most popular zone.

We also look at the number of zones used by each (sub)do-main. Figure 13a shows a CDF of the number of zones used by each subdomain. We observe that 33.2% of subdomains use only one zone, 44.5% of subdomains use two zones, and 22.3% of subdomains use three or more zones. Of the subdomains that use two or more zones, only 3.1% use zones in more than one region. Figure 13b shows the average number of zones used by the subdomains of each domain. We observe that most domains (70%) only use one zone for all subdomains; only 12% of domains use two or more zones per subdomain on average.



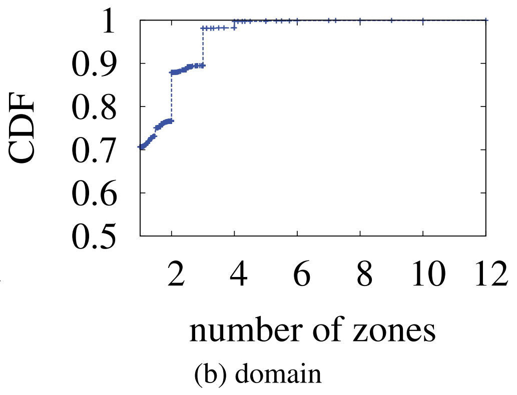


Figure 13: (a) CDF of the number of zones used by each subdomain (b) CDF of the average number of zones used by the subdomains of each domain.

Rank	domain	# subdom	# zone	*k*=1	*k*=2	*k*=3
9	amazon.com	2	4	0	0	2
13	linkedin.com	3	5	1	1	1
29	163.com	4	1	4	0	0
35	pinterest.com	18	3	10	0	8
36	fc2.com	14	5	1	11	2
38	conduit.com	1	2	0	1	0
42	ask.com	1	1	1	0	0
47	apple.com	1	1	1	0	0
48	imdb.com	2	1	2	0	0
51	hao123.com	1	1	1	0	0

Table 13: Zone usage estimates for top using zones. Column 4 is estimated total number of zones used by all subdomains. Columns 4–6 indicate the estimated number of subdomains that use k different zones.

Even for the top EC2-using domains, a large fraction of their subdomains only use a single zone (Table 13). For example, 56% of pinterest.com's EC2-using subdomains and 33% of linkedin.com's are only deployed in one zone.

Summary and implications Our two key findings in this section are that (*i*) the majority of EC2-using subdomains only use one (33.2%) or two (44.5%) zones, and (*ii*) the subdomains using a given EC2 region are not evenly spread across the availability zones in that region. The former implies that many EC2-using subdomains would be completely unavailable if a single zone failed, and many others would be severely crippled: e.g., a failure of ec2.us-east-1a would cause 16.1% of subdomains to be completely unavailable. Our later key finding implies that an outage of a particular zone in a region may have a greater negative impact than an outage of a different zone in the same region: e.g., a failure of ec2.us-east-1a would impact ≈419K subdomains, while a failure of ec2.us-east-1b would only impact ≈155K.

4.5 Wide-Area Performance and Fault Tolerance

Our results in the last section revealed that several services, even highly ranked Alexa domains, appear to use only a single region or even just a single availability zone. In this section, we explore the impact of these choices on web services' wide-area performance and tolerance to failures. We focus on EC2-using web services.

4.5.1 Wide-area Performance

The choice of region(s) by a cloud service may impact performance in at least two ways. First, clients' geo-distribution may be poorly matched to particular regions; such clients may experience poor latency and throughput compared to a more judicious deployment. Second, there could be temporary changes in which region performs best for a client due to congestion [3] or routing problems [84].

While the impact of diverse deployment of services (e.g., via CDNs) has been previously studied in other settings [51], we are unaware of any studies that assess its impact for the available diversity of modern public IaaS clouds. We therefore perform measurements to help us answer the following two questions: (*i*) To what extent does the choice of region impact performance experienced by clients of a web service? (*ii*) To what extent does the use of multiple regions (or zones) improve the client-perceived performance?

Latency measurements. To study per-region latency performance, we set up 40 m1.medium instances, 2 in each of the 20 availability zones available to us on EC2. We selected 80 geographically distributed PlanetLab [74] nodes as stand-ins for real clients and we used the hping3 utility to conduct 5 TCP pings to each of the 40 instances, from which we derive the average RTT. Pings that timed out were excluded from the calculations of the average. Probing was performed once every 15 minutes for three consecutive days.

Throughput measurements. We used the same set of 40 m1.medium EC2 instances and 80 PlanetLab nodes to measure throughput. We divided the PlanetLab nodes into two groups of 40. Each node in each group performed an HTTP get of a 2 MB file to one of the 40 EC2 instances (which were running Apache web server). At any given time, only one HTTP connection was established with each EC2 instance to avoid contention across throughput measurements. In particular, the clients in each group performed an HTTP get operation every 11.25 seconds; the download was canceled if it took more than 10 seconds. Each client accessed all 40 servers in each round, which means it took 450 seconds for one group to finish a round of downloading the file from each of the servers. So, in total, it took 15 minutes for 80 clients to perform one round of throughput measurements. The final throughput is measured as file_size / download_time. We ran the measurements for three consecutive days, for a total of 288 data points per client. The throughput measurements were intermingled with the latency measurements.

Performance across different regions. Figures 14 and 15 show the latency and throughput measurements for 15 representative PlanetLab locations and for the three US EC2 regions. The PlanetLab nodes are spread across the US and other parts of the world. We make a few key observations: (*i*) Single-region deployments must carefully choose a region. For example, the two US west regions do not offer equivalent "across-the-board" performance, with ec2.us-west-1 offering better average latency and throughput (130 ms and 1143 KB/s) than ec2.us-west-2 (145 ms

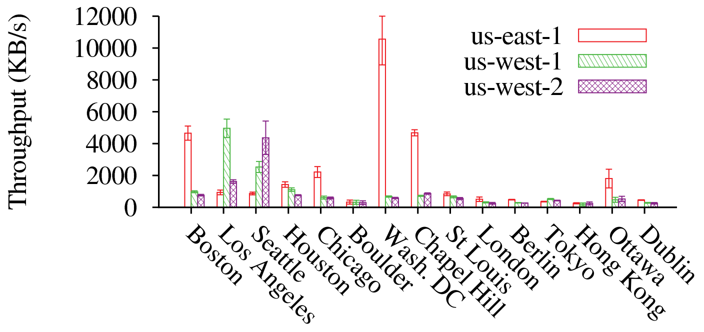


Figure 14: Average throughput between representative clients and EC2 regions in the US.

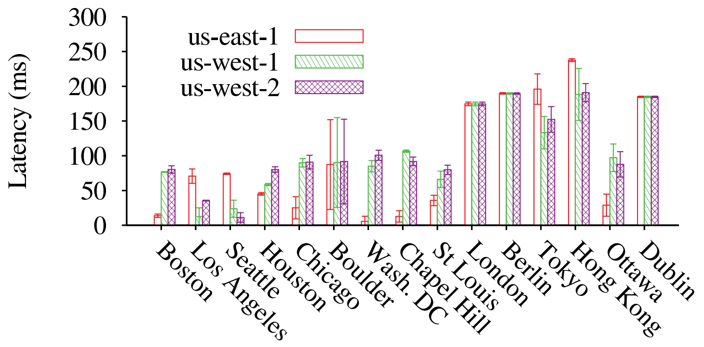
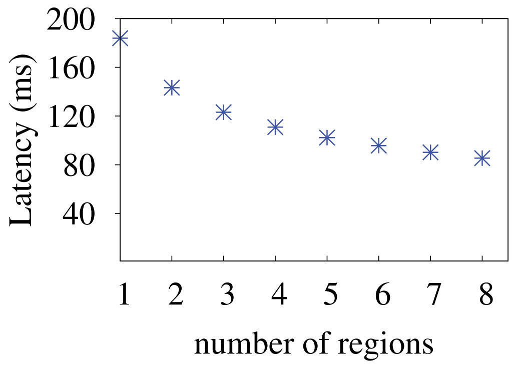


Figure 15: Average latency between representative clients and EC2 regions in the US.

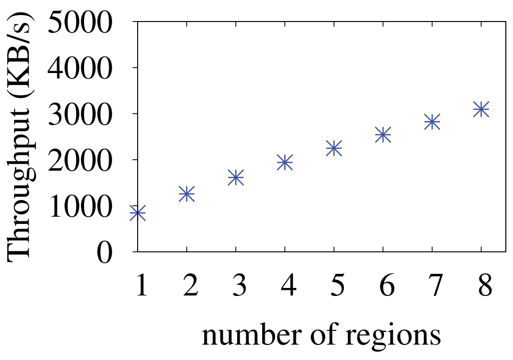
and 895 KB/s) (averaged across all client locations). (*ii*) The charts show that the region chosen to serve content to a given client can have a significant performance impact. For example, for the client in Seattle, using the ec2.us-west-2 region can reduce latency by close to a factor of 6 and improve throughput by close to a factor of 5 compared to using ec2.us-east-1. (*iii*) We also note that the way a region is chosen may depend on the client's location: always choosing ec2.us-west-1 for the Seattle client is a good idea, but for the client in Boulder, the best choice of region may change dynamically.

We now examine the relative benefits of, and choices underlying, multi-region deployments in more detail. We start by deriving an upper bound on the performance from a k-region deployment of a service. To do this, we determine the best k regions out of the 8, for $1 \leq \mathrm{k} \leq 8$. Using our measurement data we determine the overall performance that the clients would have achieved given a routing algorithm that picked the optimal region from the k for each client and at each point in time. More specifically, for each value of k we: (*i*) enumerate all size-k subsets of regions; (*ii*) for each size-k subset compute the average performance across all clients assuming each client uses the lowest latency or highest throughput region of the k at each time round (15 min); and (*iii*) choose from these the size-k subset with lowest latency or highest throughput. Figure 16 shows the results. We find that while average performance can be increased a significant amount by adding more regions to one's deployment, there is evidence of diminishing returns after $\mathrm{k} = 3$. In particular, latency decreases by 33% when $\mathrm{k} = 3$ compared to $\mathrm{k} = 1$, but only decreases by 39% when $\mathrm{k} = 4$.

We now examine what constitutes the best k regions to use. The choice of best regions, by throughput, is as follows: ec2.us-east-1 ($\mathrm{k} = 1$); ec2.us-east-1,ec2.eu-west-1 ($\mathrm{k} = 2$); ec2.us-east-1, ec2.eu-west-1, ec2.us-west-1 ($\mathrm{k} = 3$); and ec2.us-east-1, ec2.eu-west-1, ec2.us-west-1,



(a) Latency



(b) Throughput

Figure 16: Average latency/throughput across all clients using an optimal k-region deployment.

ec2.ap-southeast-1 (k $= 4$). The choice of best regions, by latency, is: ec2.us-east-1 (k $= 1$); ec2.us-east-1,ec2.ap-northeast-1 (k $= 2$); ec2.us-east-1, ec2.ap-northeast-1, ec2.us-west-1 (k $= 3$); and ec2.us-east-1, ec2.ap-northeast-1, ec2.us-west-1, ec2.ap-southeast-1 (k $= 4$).

Performance across different zones. We also investigated the difference in performance should one use different zones in the same region. We found that the zone has little impact on latency, with almost equivalent average RTTs for all clients across the two days (results omitted for brevity). For throughput, the variation appears to be somewhat higher, but not as significant as that seen across regions. We believe such variation is due to local effects, such as contention on shared instances or network switches. This is suggested as well by other recent measurements of EC2 performance variability [34]. Moreover, in the next section we show that the Internet path variability between zones is low as well.

Summary and Implications We find that using multiple regions can improve latency and throughput significantly. However, leveraging multiple regions may not be easy: while a given region always offers best performance for some clients, the choice of region for other clients will have to adapt in an online dynamic fashion. This could be achieved via global request scheduling (effective, but complex) or requesting from multiple regions in parallel (simple, but increases server load).

While a multiple region deployment helps improve web service performance and protects against major cloud failures, cloud tenants must also consider other factors in making their decision of how many and which regions to use. First, cloud providers charge for inter-region network traffic, potentially causing tenants to incur additional charges when switching to a multi-region deployment. Second, the design of particular cloud features may restrict how a tenant's data can be shared across regions: e.g., objects stored in Amazon's Simple Storage Service (S3) can only be stored in one region at a time and Amazon Machine Images (AMIs) cannot be shared between regions. Lastly, deployments that rely on fewer features may be less susceptible to failures—e.g., deployments which only use VMs, and not other services like Amazon Elastic Block Storage or Amazon ELB, have not been affected by some major outages [12, 32]—reducing the need for resiliency through the use of multiple regions.

Region	AZ1	AZ2	AZ3
ec2.us-east-1	36	36	34
ec2.us-west-1	18	19	n/a
ec2.us-west-2	19	19	19
ec2.eu-west-1	10	11	13
ec2.ap-northeast-1	9	n/a	9
ec2.ap-southeast-1	11	12	n/a
ec2.ap-southeast-2	4	4	n/a
ec2.sa-east-1	4	4	n/a

Table 14: Number of downstream ISPs for each EC2 region and zone.

4.5.2 ISP Diversity

We now investigate tolerance to wide-area faults. Having in previous sections already established the reliance of many cloud-using services on one zone or region, we now focus on diversity in the immediate downstream ISPs at each EC2 zone. Greater diversity, and an even spread of routes across downstream ISPs, generally indicates greater tolerance to failures in Internet routing.

To do this study, we set up three m1.medium instances in each of the available EC2 availability zones. Then we ran *traceroute* 50 times from each of these instances to each of 200 geographically diverse PlanetLab nodes (Figure 9). Finally, we used the UNIX 'whois' utility to determine the autonomous system (AS) number associated with the first non-EC2 hop, and we count that AS as an immediate downstream ISP for the zone hosting the instance. The discovered ASs constitute a lower bound for the true number of ASs. Table 14 gives the number of distinct ASs seen for each zone and region. We note that: (*i*) different zones in a region have (almost) the same number of downstream ISPs; and (*ii*) the extent of diversity varies across regions, with some connected to more than 30 downstream ISPs and others connected to just 4. Except for South America and Asia Pacific Sydney, other regions are well-multihomed.

We also studied the spread of routes across downstream ISPs (not shown). We found it to be rather uneven: even when using well-connected regions – e.g., ec2.us-west-1 and ec2.eu-west-1– we found that up to 31% (ec2.us-west-1) and 33% (ec2.eu-west-1) of routes use the same downstream ISP.

Summary and Implications Although individual regions are multihomed, the uneven spread of routes implies that local failures in downstream ISPs can cause availability problems for large fractions of clients of cloud-using web services. This could be overcome by using multiple regions at once, or by leveraging dynamic route control solutions [4].

5 Proposed Work

5.1 Congestion control enforcement for low latency datacenter networks

Multi-tenant datacenters are a crucial component of today's computing ecosystem. Large providers, such as Amazon, Microsoft, IBM, Google and Rackspace, support a diverse set of customers, applications and systems through their public cloud offerings. These offerings are successful in part because they provide efficient performance to a wide-class of applications running on a diverse set of platforms. Virtual Machines (VMs) play a key role in supporting this diversity by allowing customers to run applications in a wide variety of operating systems and configurations.

And while the flexibility of VMs allows customers to easily move a vast array of applications into the cloud, that same flexibility inhibits the amount of control a cloud provider can yield over VM behavior. For example, a cloud provider may be able to provide virtual networks or enforce rate limiting on a tenant VM, but it cannot control the TCP/IP stack running on the VM. As the TCP/IP stack considerably impacts overall network performance, it is unfortunate that cloud providers cannot exert a fine-grained level of control over one of the most important components in the networking stack.

Without having control over the VM TCP/IP stack, datacenter networks remain at the mercy of inefficient, out-dated or misconfigured TCP/IP stacks. TCP behavior, specifically congestion control, has been widely studied and many issues have come to light when its behavior is not optimized. For example, network congestion caused by non-optimzed stacks can lead to loss, increased latency and reduced throughput.

Thankfully, recent advances in optimizing TCP stacks for datacenter environments have shown that both high throughput and low latency can be achieved through novel TCP congestion control algorithms. Works such as DCTCP [10] and TIMELY [62] show great promise in providing high bandwidth and low latency by ensuring that network queues in switches do not fill up. And while these stacks are deployed in many of today's private datacenters [45, 81], ensuring that a vast majority of VMs within a public datacenter will update their TCP stacks to this new technology is a daunting, if not impossible task.

We plan to explore how operators can regain control of TCP's congestion control, regardless of the TCP stack running in a VM. Our aim is to allow a cloud provider to utilize advanced TCP stacks, such as DCTCP, without having control over the VM or requiring changes in network hardware. We propose implementing congestion control in the virtual switch (vSwitch) running on each server. Implementing congestion control within a vSwitch has several advantages. First, vSwitches naturally fit into datacenter network virtualization architectures and are widely deployed [73]. Second, vSwitches can easily monitor and modify traffic passing through them. Today vSwitch technology is mature and robust, allowing for a fast, scalable, and highly-available framework for regaining control over the network.

Implementing congestion control within the vSwitch has numerous challenges. First, in order to ensure adoption rates are high, the approach must work without making changes to VMs. Hypervisor-based approaches that do not modify VMs typically rely on rate limiters to limit VM traffic. Rate limiters implemented in commodity hardware do not scale in the number of flows and software implementations incur high CPU overhead [75]. Therefore, limiting a VM's TCP flows in a fine-grained, dynamic nature at scale (10,000's of flows per server [65]) with limited computational overhead remains challenging. Finally, VM TCP stacks may differ in the features they support (e.g., ECN) or the congestion control algorithm they implement, so a vSwitch congestion control implementation should work under a variety of conditions.

We propose a new technology that implements TCP congestion control within a vSwitch to help ensure VM TCP performance cannot impact the network in an adverse way. At a high-level, the vSwitch monitors all packets for a flow, modifies packets to support features not implemented in the VM's TCP stack (e.g., ECN) and reconstructs important TCP parameters for congestion control. vSwitch runs the congestion control logic specified by an administrator and then enforces an intended congestion window by modifying the receiver advertised window (`RWND`) on incoming ACKs. A policing mechanism (i.e., via dropping any excess packets not allowed by the calculated congestion window) ensures stacks cannot benefit from ignoring `RWND` and can also be used for non-TCP traffic.

5.2 Datacenter network architecture analysis and exploration

Datacenter network architecture determines how scalable the network is, how resilient it is to link or switch failures and how easily the network can be incrementally deployed. Today's practice is that network architect needs to manually infer (usually based on experiences) many key characteristics related to the candidate network topologies. So we lack a scientific and complete method to evaluate different network topologies. Therefore, there is a need to build a network architecture analysis framework to help network architect analyze and compare candidate network topologies. To compare different network topologies, we need to set up metrics to quantify different network topologies. Our first goal is to identify a set of metrics (e.g., cost, wiring complexity, bandwidth, reliability, routing convergence) that can accurately quantify datacenter network topologies. Also, we need to define a set of workloads and traffic patterns to run against the network. Given the metrics and workloads, we are very interested to answer questions like: how many hosts are disconnected when a TOR switch fails? How much bandwidth is lost when an aggregation switch fails? When a core switch fails? What about specific links?

Analyzing the existing topologies such as VL2 [37], FatTree [5], F10 [57], Jupiter [81] is our first step. Next, we want to investigate that using this analysis framework, whether we can gain insights to design better network topologies. A motivating example is F10, which identifies new striping patterns that can improve FatTree's fault-tolerance. Using our analysis framework, it will be much faster to explore new datacenter network topologies and exam the tradeoffs among different metrics for new topologies. Finally, we will investigate how we codesign routing protocol, load balancing schemes to best utilize the new topologies.

5.3 Timeline

Table 15 shows my plan for completion of the research.

Timeline	Work	Progress
	Presto & CloudMeasure	completed
Oct 2016	congestion control enforcement (NSDI'16)	proposed
Jan 2017	datacenter network architecture analysis and exploration (SIGCOMM'17)	proposed
May 2017	thesis defense	

Table 15: Plan for completion of my research

References

[1] B. Agarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese. Endre: An end-system redundancy elimination service for enterprises. In *NSDI*, 2010.

[2] K. Agarwal, C. Dixon, E. Rozner, and J. Carter. Shadow macs: Scalable label-switching for commodity ethernet. In *HotSDN*, 2014.

[3] A. Akella, S. Seshan, and A. Shaikh. An empirical evaluation of wide-area internet bottlenecks. In *IMC*, 2003.

[4] A. Akella, S. Seshan, and A. Shaikh. Multihoming performance benefits: An experimental evaluation of practical enterprise strategies. In *USENIX Annual Technical Conference, General Track*, 2004.

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

[6] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, 2010.

[7] Alexa top domains. `http://alexa.com/topsites`.

[8] Alexa web information service. `http://aws.amazon.com/awis/`.

[9] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *SIGCOMM*, 2014.

[10] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *SIGCOMM*, 2011.

[11] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *NSDI*, 2012.

[12] Amazon elb service event in the us-east region. `http://aws.amazon.com/message/680587/`.

[13] Amazon Route 53. `https://aws.amazon.com/route53/`.

[14] Amazon Web Services. `https://aws.amazon.com/`.

[15] Azure datacenter ip ranges. `http://microsoft.com/en-us/download/details.aspx?id=29840`.

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

[17] T. Benson, A. Anand, A. Akella, and M. Zhang. Microte: The case for fine-grained traffic engineering in data centers. In *CoNEXT*, 2011.

[18] C. Benvenuti. *Understanding Linux Network Internals*. " O'Reilly Media, Inc.", 2006.

[19] E. Blanton and M. Allman. On making tcp more robust to packet reordering. *ACM SIGCOMM Computer Communication Review*, 2002.

[20] S. Bohacek, J. P. Hespanha, J. Lee, C. Lim, and K. Obraczka. Tcp-pr: Tcp for persistent packet reordering. In *ICDCS*, pages 222–231, 2003.

[21] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, and D. Maltz. Per-packet load-balanced, low-latency routing for clos-based data center networks. In *CoNEXT*, 2013.

[22] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A retrospective on evolving sdn. In *HotSDN*, 2012.

[23] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, 2009.

[24] C. Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 1953.

[25] Cloud Computing Market. `http://www.cloudcomputingmarket.com/`.

[26] Cloudflare cdn. `http://cloudflare.com`.

[27] S. Das. Smart-table technology—enabling very large server, storage nodes and virtual machines to scale using flexible network infrastructure topologies. `http://www.broadcom.com/collateral/wp/StrataXGS_SmartSwitch-WP101-R.pdf`, July 2012.

[28] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the impact of packet spraying in data center networks. In *INFOCOM*, 2013.

[29] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside dropbox: understanding personal cloud storage services. In *IMC*, 2012.

[30] Ec2 public ip ranges. `https://forums.aws.amazon.com/ann.jspa?annID=1528`.

[31] Ec2 us-east disruption. `http://aws.amazon.com/message/65648/`.

[32] Ec2 us-east service event. `https://aws.amazon.com/message/680342/`.

[33] J. Edberg. Post-mortem of october 22, 2012 aws degradation. `http://techblog.netflix.com/2012/10/post-mortem-of-october-222012-aws.html`.

[34] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift. More for your money: Exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012.

[35] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing web latency: the virtue of gentle aggression. In *SIGCOMM*, 2013.

[36] Google Compute Engine. `https://cloud.google.com/compute/`.

[37] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *SIGCOMM*, 2009.

[38] L. Grossman. Large receive offload implementation in neterion 10gbe ethernet driver. In *Linux Symposium*, 2005.

[39] H. Chen and W. Song. Load Balancing without Packet Reordering in NVO3. Internet-Draft. `https://tools.ietf.org/html/draft-chen-nvo3-load-banlancing-00`, October 2014.

[40] D. R. Hanks. *Juniper QFX5100 Series: A Comprehensive Guide to Building Next-Generation Networks.* " O'Reilly Media, Inc.", 2014.

[41] K. He, A. Fisher, L. Wang, A. Gember, A. Akella, and T. Ristenpart. Next stop, the cloud: Understanding modern web service deployment in ec2 and azure. In *IMC*, 2013.

[42] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. Presto: Edge-based load balancing for fast datacenter networks. In *SIGCOMM*, 2015.

[43] IBM Bluemix. `http://www.ibm.com/cloud-computing/bluemix/`.

[44] IBM SoftLayer. `http://www.softlayer.com/`.

[45] G. Judd. Attaining the promise and avoiding the pitfalls of tcp in the datacenter. In *NSDI*, 2015.

[46] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene. Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *CoNEXT*, 2014.

[47] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: Measurements & analysis. In *IMC*, 2009.

[48] R. Kapoor, A. C. Snoeren, G. M. Voelker, and G. Porter. Bullet trains: A study of nic burst behavior at microsecond timescales. In *CoNEXT*, 2013.

[49] Knock. `http://code.google.com/p/knock`.

[50] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, et al. Network virtualization in multi-tenant datacenters. In *NSDI*, 2014.

[51] R. Krishnan, H. V. Madhyastha, S. Srinivasan, S. Jain, A. Krishnamurthy, T. Anderson, and J. Gao. Moving beyond end-to-end path information to optimize cdn performance. In *IMC*, 2009.

[52] C. Labovitz. How big is amazon's cloud? `http://deepfield.net/blog`, 2012.

[53] C. Lee, C. Park, K. Jang, S. Moon, and D. Han. Accurate latency-based congestion feedback for datacenters. In *USENIX ATC*, volume 15, 2015.

[54] K.-C. Leung, V. O. Li, and D. Yang. An overview of packet reordering in transmission control protocol (tcp): Problems, solutions, and challenges. *Parallel and Distributed Systems, IEEE Transactions on*, 2007.

[55] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: comparing public cloud providers. In *IMC*, 2010.

[56] A. Li, X. Zong, S. Kandula, X. Yang, and M. Zhang. Cloudprophet: towards application performance prediction in cloud. In *ACM SIGCOMM Computer Communication Review*, 2011.

[57] V. Liu, D. Halperin, A. Krishnamurthy, and T. E. Anderson. F10: A fault-tolerant engineered network. In *NSDI*, 2013.

[58] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. Technical report, 2014.

[59] R. McMillan. Amazon's secretive cloud carries 1 percent of the internet. `http://www.wired.com/wiredenterprise/2012/04/amazon-cloud/`, 2012.

[60] A. Menon and W. Zwaenepoel. Optimizing tcp receive performance. In *USENIX Annual Technical Conference*, 2008.

[61] Microsoft Azure. `https://azure.microsoft.com/en-us/?b=16.33`.

[62] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats, et al. Timely: Rtt-based congestion control for the datacenter. In *SIGCOMM*, 2015.

[63] J. C. Mogul and K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 1997.

[64] T. P. Morgan. A rare peek into the massive scale of aws. `http://www.enterprisetech.com/2014/11/14/rare-peek-massive-scale-aws/`, 2014.

[65] M. Moshref, M. Yu, A. Sharma, and R. Govindan. vcrib: Virtualized rule management in the cloud. In *NSDI*, 2013.

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

[67] Passive dns network mapper. `http://code.google.com/p/dnsmap/`.

[68] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, et al. Ananta: Cloud scale load balancing. In *SIGCOMM*, 2013.

[69] V. Paxson. End-to-end Internet Packet Dynamics. In *ACM SIGCOMM Computer Communication Review*, 1997.

[70] V. Paxson. Bro: A system for detecting detwork intruders in real-time. In *USENIX Security Symposium (SSYM)*, 1998.

[71] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized zero-queue datacenter network. In *SIGCOMM*, 2014.

[72] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker. Extending networking into the virtualization layer. In *Hotnets*, 2009.

[73] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, et al. The design and implementation of open vswitch. In *NSDI*, 2015.

[74] Planetlab. `http://planet-lab.org/`.

[75] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. SENIC: Scalable NIC for End-host Rate Limiting. In *NSDI*, 2014.

[76] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath tcp. In *SIGCOMM*, 2011.

[77] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: millisecond-scale monitoring and control for commodity networks. In *SIGCOMM*, 2014.

[78] Receive segment coalescing (rsc). `http://technet.microsoft.com/en-us/library/hh997024.aspx`.

[79] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[80] J. Schectman. Netflix amazon outage shows any company can fail. `http://blogs.wsj.com/cio/2012/12/27/netflix-amazon-outage-shows-any-company-can-fail/`, 2012.

[81] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM*, 2015.

[82] S. Sinha, S. Kandula, and D. Katabi. Harnessing tcps burstiness using flowlet switching. In *HotNets*, 2004.

[83] B. Stephens, A. L. Cox, A. Singla, J. Carter, C. Dixon, and W. Felter. Practical dcb for improved data center networks. In *INFOCOM*, 2014.

[84] R. Teixeira, A. Shaikh, T. Griffin, and J. Rexford. Dynamics of hot-potato routing in ip networks. In *ACM SIGMETRICS Performance Evaluation Review*, 2004.

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

[86] H. Wu, Z. Feng, C. Guo, and Y. Zhang. Ictcp: incast congestion control for tcp in data-center networks. *IEEE/ACM Transactions on Networking (TON)*, 2013.

[87] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. Detail: Reducing the flow completion time tail in datacenter networks. In *SIGCOMM*, 2012.

[88] M. Zhang, B. Karp, S. Floyd, and L. Peterson. Rr-tcp: A reordering-robust tcp with dsack. In *ICNP*, 2003.

[89] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat. Wcmp: Weighted cost multipathing for improved fairness in data centers. In *EuroSys*, 2014.

[90] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale rdma deployments. In *SIGCOMM*, 2015.