SEATTLE: A Scalable Ethernet Architecture for Large Enterprises

CHANGHOON KIM, Microsoft
MATTHEW CAESAR, University of Illinois, Urbana-Champaign
JENNIFER REXFORD, Princeton University

IP networks today require massive effort to configure and manage. Ethernet is vastly simpler to manage, but does not scale beyond small local area networks. This article describes an alternative network architecture called SEATTLE that achieves the best of both worlds: The scalability of IP combined with the simplicity of Ethernet. SEATTLE provides plug-and-play functionality via flat addressing, while ensuring scalability and efficiency through shortest-path routing and hash-based resolution of host information. In contrast to previous work on identity-based routing, SEATTLE ensures path predictability, controllability, and stability, thus simplifying key network-management operations, such as capacity planning, traffic engineering, and troubleshooting. We performed a simulation study driven by real-world traffic traces and network topologies, and used Emulab to evaluate a prototype of our design based on the Click and XORP open-source routing platforms. Our experiments show that SEATTLE efficiently handles network failures and host mobility, while reducing control overhead and state requirements by roughly two orders of magnitude compared with Ethernet bridging.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Network**]: Network Architecture and Design; C.2.2 [**Computer-Communication Network**]: Network Protocols; C.2.5 [**Computer-Communication Network**]: Local and Wide-Area Networks

General Terms: Design, Experimentation, Management

Additional Key Words and Phrases: Enterprise network, data-center network, routing, scalability, ethernet

ACM Reference Format:
Kim, C., Caesar, M., and Rexford, J. 2011. SEATTLE: A scalable Ethernet architecture for large enterprises. ACM Trans. Comput. Syst. 29, 1, Article 1 (February 2011), 35 pages.
DOI = 10.1145/1925109.1925110 http://doi.acm.org/10.1145/1925109.1925110

1. INTRODUCTION

Ethernet stands as one of the most widely used networking technologies today. Due to its simplicity and ease of configuration, many enterprise, access-provider, and data-center networks utilize Ethernet as an elementary building block. Each host in an Ethernet is assigned a persistent and unique MAC address, and Ethernet bridges automatically learn host addresses and locations. These "plug-and-play" semantics simplify many critical aspects of network configuration. Meanwhile, flat addressing simplifies the handling of both host-location and network-topology changes, obviating the need for network administrators to reassign addresses.

Authors' addresses: C. Kim, One Microsoft Way, Redmond, WA 98052; email: kim.changhoon@gmail.com; M. Caesar, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801; email: Caesar@ca.illinois.edu; J. Rexford, Department of Computer Science, Princeton University, 25 Olden Street, Princeton, NJ 08540; email: jrex@cs.princeton.edu.

DOI 10.1145/1925109.1925110 http://doi.acm.org/10.1145/1925109.1925110

However, Ethernet is facing revolutionary challenges. Today's layer-2 networks are being built at an unprecedented size and with highly demanding requirements in terms of efficiency, scalability, and availability. Large data centers are being built, comprising hundreds of thousands of physical and virtual machines within a single facility [Arregoces and Portolani 2003; Barroso and Holzle 2009], and maintained by hundreds of network operators. To increase machine utilization, facilitate maintenance, and reduce operational cost, these data centers employ various agility-enhancing mechanisms such as live virtual-machine migration, fast machine re-imaging (i.e., non-live machine migration), and dynamic adjustment of resource shares (e.g., expanding or shrinking the size of a machine pool running a distributed application). All these agility mechanisms place additional requirements on handling very high rates of host and network churn—host arrival and departure, host address and location changes, IP subnet and Ethernet Virtual LAN (VLAN) re-configuration, etc. In particular, cloud-service data centers face especially challenging requirements, as they must offer networking service for a large number of tenants (cloud-service customers sharing the same data center) whose arrival and departure rates can be extremely high owing to the usage-based charging policy and the machine-independent, transient nature of jobs. Meanwhile, large metro Ethernet deployments easily contain over a million hosts and tens of thousands of bridges [Halabi 2003]. Ethernet is also being increasingly deployed in highly dynamic environments, such as backhaul for wireless campus networks, and as transport for developing regions [Hudson 2002].

Ethernet becomes all the more important in these environments because it allows hosts to retain their IP addresses as long as they move within a single layer-2 domain (i.e., IP subnet). This property is highly useful for ensuring service continuity across host-location changes as well as simplifying both network and host configuration related to policy enforcement (e.g., access control). Despite these benefits, conventional Ethernet has some critical limitations. First, Ethernet bridging relies on network-wide *flooding* to locate end hosts. This results in large overhead to disseminate and store host state that grows with the size of the network. Second, Ethernet forces paths to comprise a *spanning tree*. Spanning trees perform well for small networks that often do not have many redundant paths anyway, but introduce substantial inefficiencies on larger networks that have more demanding requirements for low latency, high availability, and traffic engineering. Finally, critical bootstrapping protocols used frequently by end hosts, such as Address Resolution Protocol (ARP) and Dynamic Host Configuration Protocol (DHCP), rely on *broadcasting*. Not only does broadcasting waste useful network and end-host resources, doing so additionally introduces security vulnerabilities and privacy concerns.

Network administrators sidestep Ethernet's inefficiencies today by interconnecting small Ethernet LANs using routers running the Internet Protocol (IP). IP routing ensures efficient and flexible use of networking resources via shortest-path routing. It also has control overhead and forwarding-table sizes that are proportional to the number of subnets (i.e., prefixes), rather than the number of hosts. However, introducing IP routing breaks many of the desirable properties of Ethernet. For example, network administrators must now subdivide their address space to assign IP prefixes across the topology, and update these configurations when the network-design changes. Subnetting leads to wasted address space, and laborious configuration tasks. Although DHCP automates host address configuration, maintaining consistency between DHCP servers and routers still remains challenging. Moreover, since IP addresses are not persistent identifiers, ensuring service continuity across location changes (e.g., due to virtual machine migration or physical mobility) becomes more challenging. Additionally, access-control policies must be specified based on the host's current position, and updated when the host moves.

Alternatively, operators may use VLANs, which allow administrators to build IP subnets irrespective of hosts' location. Hence, by provisioning each VLAN over a large fraction of—if not an entire—network, administrators can lower the overhead of address and access-control policy re-configuration due to host mobility. Unfortunately, however, having large VLANs counteracts the benefits of broadcast scoping, and worsens data-plane efficiency, as a larger spanning tree is used in each VLAN to forward traffic. In addition, properly determining the coverage of a VLAN (i.e., deciding which bridges and links participate in a VLAN) requires precise knowledge about hosts' communication and mobility patterns and thus is extremely hard to automate. Moreover, since hosts in different VLANs still require IP to communicate with one another, this architecture still inherits many of the challenges of IP mentioned above, such as address-space fragmentation.

In this article, we address the following question: Is it possible to build a protocol that maintains the same configuration-free properties as Ethernet bridging, yet scales to large dynamic networks? To answer, we present a Scalable Ethernet Architecture for Large Enterprises (SEATTLE). Specifically, SEATTLE offers the following features:

A One-Hop, Network-Layer DHT. SEATTLE forwards packets based on end-host MAC addresses. However, SEATTLE does not require each switch to maintain state for every host, *nor* does it require network-wide floods to disseminate host locations. Instead, SEATTLE uses the global switch-level view provided by a link-state routing protocol to form a one-hop DHT [Gupta et al. 2004], which stores the location of each host. We also use this network-layer DHT to build a flexible directory service which enables address resolution (e.g., storing the MAC address associated with an IP address) as well as convenient service discovery (e.g., maintaining DHCP server addresses, the least loaded DNS server's address, or a printer within the domain). In addition, to reduce lookup latency and enable fault isolation in a large network deployed over a wide area, we present a hierarchical configuration of multiple regional DHTs.

Traffic-Driven Location Resolution and Caching. To forward packets along shortest paths and to avoid excessive load on the directory service, switches can cache responses to queries. Caching routing information is a well studied topic especially for Internet routers which have to maintain a large amount of routing information [Jain and Routhier 1986; Jain 1990; Feldmeier 1988; Heimlich 1990; Partridge 1996; Partridge et al. 1998; Kim et al. 2009]. The route-caching design we propose is particularly effective for the target operational environment we envision—enterprises, metro-area, and data-center networks. In these networks, many hosts typically communicate only with a small number of other hosts (e.g., web, mail, or proxy servers) which are commonly popular across the whole network [Aiello et al. 2005].[1] Hence, SEATTLE switches can achieve a very high cache-hit ratio by maintaining the information about the small working set of destination hosts. Our route-caching design also employs a unique mechanism that addresses the key limitation of the earlier route-caching work, namely slow-path forwarding upon cache misses. Unlike this earlier design, a

[1]Special-purpose cluster or data-center networks that are predominantly used to run data-parallel distributed computing [Dean and Ghemawat 2004; Isard et al. 2007] or high-performance computing applications might be an exception to this. In such a network, a host can communicate with a large number of other hosts in a short period of time. In fact, one of the primary goals of the job scheduler in such a system is avoiding a skewed host-popularity distribution, making SEATTLE's host-information caching less effective. Nonetheless, our SEATTLE design, even without host-information caching, can still ensure most of the core benefits (e.g., smaller forwarding tables, fewer control-message exchanges, zero flood) over conventional Ethernet, and we specifically demonstrate some of these benefits in Section 6. Furthermore, in data-center networks where latency and workload increase due to a longer stretch is tolerable, random traffic indirection via the network-layer DHT can offer unique benefits, such as traffic-oblivious load spreading [Kodialam et al. 2004].

SEATTLE switch can simply forward a packet to another switch chosen by a simple hash function even when the packet causes a cache miss, completing the entire packet-forwarding process on the fast path. Furthermore, SEATTLE also provides a way to piggyback location information on ARP replies, which eliminates the need for separate location resolution when forwarding data packets following the ARP resolution. All these mechanisms allow data packets to directly traverse the shortest path, making the network's forwarding behavior predictable and stable as well as simplifying traffic engineering and network troubleshooting. This is one of the core benefits of SEATTLE compared to conventional DHT-based networking systems.

A Scalable, Prompt Cache-Update Protocol. Unlike Ethernet which relies on timeouts or broadcasts to keep forwarding tables up-to-date, SEATTLE proposes an explicit and reactive cache update protocol that leverages only unicast. This ensures that all packets are delivered based on up-to-date state, ensuring eventual consistency with much lower control-plane overhead compared to Ethernet bridging and other DHT systems. Further, in contrast to conventional DHTs, this state-update process is directly triggered by network-layer changes (specifically link-state advertisements). This enables fast and accurate reaction to host- and network-state changes without requiring any additional monitoring protocols. For example, by observing link-state advertisements, switches can precisely determine when a switch is no longer reachable and accordingly evict all hosts' entries that are no longer valid. Through similar approaches, SEATTLE also seamlessly supports host-information (e.g., location, name, address) changes.

Despite these features, our design remains compatible with existing applications and protocols running at end hosts, as well as all Ethernet addressing modes (unicast, multicast, and broadcast). For example, SEATTLE allows hosts to generate broadcast ARP or DHCP messages and internally converts them into unicast queries to a directory service. SEATTLE switches can also handle general (i.e., non-ARP and non-DHCP) broadcast traffic through loop-free multicasting. To offer broadcast scoping and access control, SEATTLE also provides a more scalable and flexible mechanism that allows administrators to create VLANs without requiring complicated VLAN-trunk configuration.

The roadmap of this article is as follows. We first motivate our work in Section 2 by identifying the shortcomings of the conventional enterprise-network technologies. Then we describe our main contributions in Sections 3 and 4 where we introduce a very simple yet highly scalable directory service that enables shortest-path forwarding while maintaining the same semantics as Ethernet. In Section 5, we further improve the SEATTLE design by proposing a mechanism that ensures backwards-compatibility with conventional Ethernet. We then evaluate our protocol using large-scale packet-level simulations in Section 6. To test our solution for real networks, we also implement a prototype switch using open-source routing platforms and summarize the prototype switch design and architecture in Section 7. Then, we perform emulation tests using the prototype and present evaluation results in Section 8. Our results presented in Sections 6 and 8 show that SEATTLE scales to networks containing two orders of magnitude more hosts than a traditional Ethernet network. As compared with other flat-networking designs, SEATTLE reduces state requirements required to achieve reasonably low stretch by a factor of ten, and improves path stability by more than three orders of magnitude under typical workloads. SEATTLE also handles network topology changes and host mobility without significantly increasing control overhead. Finally, we compare in Section 9 our SEATTLE work with related network designs, including advanced Ethernet architectures (e.g., RBridges [Perlman 2004], IETF TRILL [TRILL 2010], SmartBridges [Rodeheffer et al. 2000]), flat networking

architectures (e.g., ROFL [Caesar et al. 2006b], VRR [Caesar et al. 2006a], UIP [Ford 2004]), and data-center network architectures (e.g., VL2 [Greenberg et al. 2009], PortLand [Mysore et al. 2009]).

2. TODAY'S ENTERPRISE AND ACCESS NETWORKS

To provide background for the remainder of this article, and to motivate SEATTLE, this section explains why Ethernet bridging does not scale. Then, we describe hybrid IP/Ethernet networks and VLANs, two widely used approaches which improve scalability over conventional Ethernet, but introduce management complexity, eliminating the "plug-and-play" advantages of Ethernet.

2.1 Ethernet Bridging

An Ethernet network is composed of *segments*, each comprising a single physical layer.[2] Ethernet *bridges* are used to interconnect multiple segments into a multi-hop network, namely a LAN, forming a single *broadcast domain* [Varghese and Perlman 1990]. Each host is assigned a unique 48-bit MAC (Media Access Control) address. A bridge learns how to reach hosts by inspecting the incoming frames, and associating the source MAC address with the incoming port. A bridge stores this information in a *forwarding table* that it uses to forward frames toward their destinations. If the destination MAC address is not present in the forwarding table, the bridge sends the frame on all outgoing ports, initiating a domain-wide flood. Bridges also flood frames that are destined to a broadcast MAC address. Since Ethernet frames do not carry a TTL (Time-To-Live) value, the existence of a loop in the topology can lead to *broadcast storms*, where frames are repeatedly replicated and forwarded along the loop. To avoid this, bridges in a broadcast domain coordinate to compute a *spanning tree* [Perlman 1985]. Administrators first select and configure a single *root bridge*, and then the bridges collectively compute a spanning tree based on distances to the root. Unfortunately, Ethernet-bridged networks cannot grow to a large size due to the following reasons.

Globally Disseminating Every Host's Location. Flooding and source-learning introduce two problems in a large broadcast domain. First, the forwarding table at a bridge can grow very large because flat addressing increases the table size in proportion to the total number of hosts in the network. Second, the control overhead required to disseminate each host's information via flooding can be very large, wasting link bandwidth and processing resources. Since hosts (or their network interfaces) power up/down (manually, or dynamically to reduce power consumption), and change location relatively frequently, flooding is an expensive way to keep per-host information up-to-date. Moreover, malicious hosts can intentionally trigger repeated network-wide floods through, for example, MAC address scanning attacks [Allman et al. 2007].

Inflexible Route Selection. Forcing all traffic to traverse a single spanning tree makes forwarding more failure-prone and leads to suboptimal paths and uneven link loads [Perlman 1999]. Load is especially high on links near the root bridge. Thus, choosing the right root bridge is extremely important, imposing an additional administrative burden. Moreover, using a single tree for all communicating pairs, rather than shortest paths, significantly reduces the aggregate throughput of a network because links not present in the tree cannot carry any traffic.

[2]Most enterprises and data centers today build switched Ethernet networks, rather than shared-medium Ethernet networks. In a switched Ethernet network, a segment is a point-to-point link connecting an end host and a bridge, or a pair of bridges. SEATTLE can work with both types of links because it leverages a link-state routing protocol, which supports both switched and shared-medium links.

Dependence on Broadcasting for Basic Operations. DHCP [Droms 1997] and ARP [Plummer 1982] are used to assign IP addresses and manage mappings between MAC and IP addresses, respectively. A host broadcasts a DHCP-discovery message whenever it believes its network attachment point has changed. Broadcast ARP requests are generated more frequently, whenever a host needs to know the MAC address associated with the IP address of another host in the same broadcast domain. Relying on broadcast for these operations degrades network performance. Moreover, every broadcast message must be processed by every end host; since handling of broadcast frames is often application or OS-specific, these frames are not filtered by the network interface card, and instead must interrupt the CPU. For portable devices on low-bandwidth wireless links, receiving ARP packets can consume a significant fraction of the available bandwidth, processing, and power resources. Moreover, the use of broadcasting for ARP and DHCP opens vulnerabilities to malicious hosts that can easily launch ARP or DHCP floods [Myers et al. 2004].

2.2 Hybrid IP/Ethernet Architecture

One way of dealing with Ethernet's limited scalability is to build enterprise and access provider networks out of multiple LANs interconnected by *IP routing*. In these hybrid networks, each LAN contains at most a few hundred hosts that collectively form an *IP subnet*. Communication across subnets is handled via certain fixed nodes called *default gateways*. Each IP subnet is allocated an *IP prefix*, and each host in the subnet is then assigned an IP address from the subnet's prefix. Unlike a MAC address, which functions as a host identifier, an IP address denotes the host's current location in the network.

The biggest problem of the hybrid architecture is its configuration overhead. Configuring hybrid networks today represents an enormous challenge. Some estimates put 70% of an enterprise network's operating cost as maintenance and configuration, as opposed to equipment costs or power usage [Kerravala 2002]. In addition, involving human administrators in the loop increases reaction time to faults and increases potential for misconfiguration.

Configuration Overhead due to Hierarchical Addressing. An IP router cannot function correctly until administrators specify subnets on router interfaces, and direct routing protocols to advertise the subnets. Similarly, an end host cannot access the network until it is configured with an IP address corresponding to the subnet where the host is currently located. Assigning IP prefixes to subnets, and associating subnets with router interfaces is typically a manual process, as the assignment must follow the addressing hierarchy, yet must reduce wasted addresses, and must consider future use of addresses to minimize later reassignment. Despite automating end-host configuration, DHCP introduces additional configuration overhead for managing the DHCP servers. In particular, maintaining consistency between routers' subnet configuration and DHCP servers' address allocation configuration, or coordination across distributed DHCP servers are not simple. Finally, network administrators must continually revise this configuration to handle network changes.

Complexity in Implementing Networking Policies. Administrators today use a collection of access controls, QoS (Quality of Service) controls [King 2004], and other policies to control the way packets flow through their networks. These policies are typically defined based on IP prefixes. However, since prefixes are assigned based on the topology, changes to the network design require these policies to be rewritten. More significantly, rewriting networking policies must happen immediately after the network design changes to prevent reachability and performance problems from

happening, as well as to avoid vulnerabilities. Ideally, administrators should only need to update policy configurations when the policy itself, not the network, changes.

Limited Mobility Support. Supporting seamless host mobility is becoming increasingly important in various networks. To improve power efficiency by adapting to workload and to minimize service disruption during maintenance operations, data-center administrators often relocate virtual machines (either via machine re-imaging or live migration). Large universities or enterprises often build campus-wide wireless networks, using a wired backhaul to support host mobility across access points. To ensure service continuity and minimize the policy-update overhead mentioned above, it is highly desirable for a host to retain its IP address regardless of its location in these networks. Unfortunately, hybrid networks constrain host mobility only within a single, usually small, subnet. In a data center, this can interfere with the ability to handle load spikes seamlessly; in wireless backhaul networks, this can cause service disruptions. One way to deal with this is to increase the size of subnets by increasing broadcast domains, introducing the scaling problems mentioned in Section 2.1.

2.3 Virtual LANs

VLANs address some of the problems of Ethernet and IP networks. VLANs allow administrators to group multiple hosts sharing common networking policies into a single broadcast domain. Unlike a physical LAN, a VLAN can be defined logically, regardless of individual hosts' locations in a network. VLANs can also be overlapped by allowing bridges (not hosts) to be configured with multiple VLANs. By dividing a large bridged network into several appropriately-sized VLANs, administrators can reduce the broadcast overhead imposed on hosts in each VLAN and also ensure isolation among different host groups. Compared with IP, VLANs simplify mobility because hosts can retain their IP addresses while moving between bridges in the same VLAN. This also reduces policy reconfiguration overhead. Despite these benefits, however, VLANs have the following limitations:

Trunk Configuration Overhead. Extending a VLAN across multiple bridges requires the VLAN to be trunked (provisioned) at each of the bridges participating in the VLAN. Deciding which bridges should participate in a given VLAN must consider traffic and host-mobility patterns to ensure efficiency and hence is often done manually.

Limited Control-Plane Scalability. Although VLANs reduce the broadcast overhead imposed on a particular end host, bridges provisioned with multiple VLANs must maintain forwarding-table entries and process broadcast traffic for every active host in *every* VLAN configured on themselves. Unfortunately, to enhance resource utilization and host mobility, and to reduce trunk configuration overhead, VLANs are often provisioned larger than necessary, worsening this problem. A large forwarding table complicates bridge design, since forwarding tables in Ethernet bridges are typically implemented using Content-Addressable Memory (CAM) or off-chip SRAM (Static Random Access Memory), an expensive and power-intensive technology.

Insufficient Data-Plane Efficiency. Forwarding traffic along a single spanning tree in a VLAN prevents certain links from being used. Since larger enterprises and data centers often have richer topologies for greater reliability and performance, this limitation gets much more pronounced. Although configuring a disjoint spanning tree for each VLAN [IEEE 802.1Q 2005; Sharma et al. 2004] may improve load balance and increase aggregate throughput, effective use of per-VLAN trees requires periodically moving the roots and re-balancing the trees, which must be manually updated as

traffic shifts. Moreover, inter-VLAN traffic must be routed via IP gateways, rather than shortest physical paths.

3. NETWORK-LAYER ONE-HOP DHT

The goal of a networking system is to deliver packets or frames to a destination specified by an address. To do this, network devices collectively provide end hosts with a service that maps destination addresses (or groups of addresses) to the physical locations of the hosts owning the addresses. Each network device implements this service by maintaining next-hop pointers associated with host addresses in its forwarding table and relies on a routing mechanism (e.g., link-state routing protocols in case of IP, or domain-wide flooding and MAC learning in case of Ethernet) to keep these pointers up to date.

In order to provide the same semantics to end hosts as those conventional networking systems and yet scale to a large size, SEATTLE introduces a *distributed directory service* that maintains mappings between hosts' addresses, locations, and various other types of information (e.g., names, identifiers). Specifically, the directory service is built using a *one-hop network-layer DHT*.

We use a *one-hop* DHT to reduce lookup latency and complexity, as well as simplify certain aspects of network administration such as traffic engineering and troubleshooting. We use a *network-layer* approach that stores mappings at switches, rather than at separate management servers such as Ethane servers [Casado et al. 2009], for several critical reasons. First, our network-layer approach ensures fast, efficient, and accurate reaction to host and network events (e.g., host/switch failures and recoveries, host location/address changes) because switches can easily learn about those events simply by monitoring link-state advertisements, by monitoring the liveness of host-facing ports, or by snooping DHCP messages, without requiring any additional control-plane protocol. Additionally, our network-layer approach can obviate large buffers at ingress switches needed to store packets during host-information resolution, because ingress switches can simply forward the packets to other switches that are known to keep valid host information. While it is also conceivable to use the same approach with separate management servers, then those management servers have to be equipped with enough data-plane capacity to cope with real data packets, rather than handle a small volume of resolution messages. Meanwhile, our network-layer approach also allows storage capacity to increase naturally with network size, helping the administrators avoid critical operational concerns regarding when, where, and how many external management servers they should deploy.

While we primarily focus in this article on using SEATTLE's distributed directory service as a mechanism to enable large-scale Ethernet networks, we also envision various additional uses of the distributed directory service to maintain other kinds of key-value pairs in a highly scalable and eventually consistent fashion. To demonstrate this generality, we show how we can use the distributed directory service to implement a few different protocols' semantics, such as Ethernet bridging (in Section 4.1), ARP (in Section 4.2), DHCP (in Section 5.1), and even a general service discovery protocol that can replace any application-specific broadcast (in Section 3.1.3).

3.1 A Highly Scalable, Multi-Purpose Key-Value Store Built with a One-Hop DHT

Our distributed directory has two main parts. First, switches run a link-state protocol. This ensures that each switch can observe all other switches in the network, offering a foundation for building an eventually-consistent distributed directory. Additionally, this allows any switch to forward traffic to any other switch along shortest paths. Second, switches use a *common hash function* to map host information to a switch.

Each host-information entry is maintained in the form of (*key, value*). Examples of these key-value pairs are (*MAC address, location*), and (*IP address, MAC address*).

3.1.1 Link-State Protocol Maintaining Switch State and Topology. SEATTLE enables shortest-path forwarding by running a link-state protocol. However, distributing end-host information in link-state advertisements, as advocated in previous proposals [Myers et al. 2004; Perlman 2004; Rodeheffer et al. 2000; TRILL 2010], would lead to serious scaling problems in the large networks we consider. Instead, SEATTLE's link-state protocol maintains only the *switch-level* topology, which is much more compact and stable. SEATTLE switches use this link-state information to compute shortest paths for unicasting as well as multicast trees for multicasting and broadcasting.

To automate configuration of the link-state protocol, SEATTLE switches run a discovery protocol to determine which of their links are attached to hosts and which are connected to other switches. Distinguishing between these different kinds of links is done by sending control messages that hosts ignore.[3] To identify themselves in the link-state protocol, SEATTLE switches determine their own unique *switch IDs* without administrator involvement and announce those IDs via link-state advertisements. For example, each switch does this by choosing the MAC address of one of its interfaces as its switch ID.

3.1.2 Hashing Key-Value Pairs onto Switches. Instead of disseminating per-host information in link-state advertisements, SEATTLE switches maintain (i.e., publish, store, and retrieve) this information in an on-demand fashion via a simple hashing mechanism. This information is stored in the form of (key = k, value = v) pairs. A publisher switch s_a wishing to publish a (k, v) pair to the directory service uses a hash function $\mathcal{F}$, which maps k to a switch identifier $\mathcal{F}(k) = r_k$ and instructs switch r_k to store the mapping (k, v). We refer to r_k as the *resolver* for k. A different switch s_b may then look up the value associated with k by using the same hash function to identify which switch is k's resolver. This works because each switch knows all the other switches' identifiers via link-state advertisements from the routing protocol, and hence $\mathcal{F}$ works identically for all switches. Switch s_b may then forward a lookup request to r_k to retrieve the value v. Switch s_b may optionally cache the result of its lookup, to reduce redundant resolutions. All control messages, including lookup and publish messages, are unicast with reliable delivery.

Reducing Control Overhead with Consistent Hashing. When the set of switches changes due to a network failure or recovery, some keys have to be rehashed to different resolver switches. To minimize this rehashing overhead, SEATTLE utilizes *Consistent Hashing* [Karger et al. 1997] for $\mathcal{F}$. This mechanism is illustrated in Figure 1. A consistent hashing function maps keys to bins such that the change of the bin set causes minimal churn in the mapping of keys to bins. In SEATTLE, each switch corresponds to a bin, and a host's information corresponds to a key. Formally, given a set $S = \{s_1, s_2, ..., s_n\}$ of switch identifiers, and a key k,

$$\mathcal{F}(k) = argmin_{\forall s_i \in S}\{\mathcal{D}(\mathcal{H}(k), \mathcal{H}(s_i))\}$$

where $\mathcal{H}$ is a regular hash function, and $\mathcal{D}(x, y)$ is a simple metric function computing the counter-clockwise distance from x to y on the circular hash-space of $\mathcal{H}$. This means $\mathcal{F}$ maps a key to a switch in such a way that the hash of the switch's id does not exceed the hash of the key on the hash space of $\mathcal{H}$. As an optimization to improve resilience to failures, a key may be additionally mapped to the next m closest switches along the

[3]This process is similar to how Ethernet bridges distinguish switches from hosts when building a spanning tree.

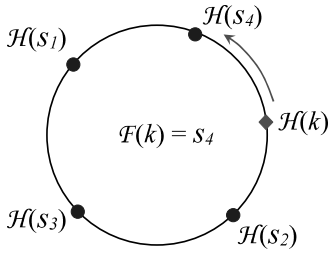


Fig. 1. Keys are consistently hashed onto resolver switches (s_i).

hash ring. However, in our evaluation, we will assume this optimization is disabled by default. The use of consistent hashing ensures that on average, a single switch failure or recovery triggers $|N|/|S|$ key rehashing events, where $|N|$ is the number of keys (i.e., the number of end hosts in our usage) and $|S|$ is the number of switches in the network. That is, in a network with 100,000 end hosts and 2,000 switches, a single switch failure or recovery generates only 50 key-value re-publication messages on average.

Balancing Load with Virtual Switches. The scheme described so far assumes that all switches are equally powerful, and hence low-end switches will need to cope with the same directory-service workload as more powerful switches do. In practice, however, the capability of adjusting the workload is much needed for network-management purposes. Hence, we propose an extension scheme based on running multiple virtual switches on each physical switch. A single switch locally creates one or more virtual switches. The switch may then increase or decrease its load by spawning/destroying these virtual switches. Unlike techniques used in traditional DHTs for load balancing [Dabek et al. 2001], it is *not* necessary for each of our virtual switches to be individually advertised to other physical switches, substantially reducing the size of link-state advertisements. Specifically, instead of advertising every virtual switch in link-state advertisement, a SEATTLE switch only advertises the number of virtual switches it is currently running. Upon receiving that number, other switches locally generate the virtual switch IDs of the physical switch.[4] Note that it is possible to automate determining a desirable number of virtual switches per physical switch [Godfrey et al. 2003].

3.1.3 Enabling Flexible Service Discovery. This design also enables more flexible service discovery mechanisms without the need to perform network-wide broadcasts. This is done by utilizing the hash function $\mathcal{F}$ to map a string defining the service to a switch. For example, a printer may hash the string *"PRINTER"* and map the hash to a switch, at which the printer may store its location or address information. Other switches can then reach the printer using the hash of the pre-shared string. Services may also encode additional attributes, such as load or network location, as simple extensions. Multiple servers can redundantly register themselves with a common string to implement anycasting, implementing anycast or multicast. Services can be named and described using techniques shown in previous work [Adjie-Winoto et al. 1999].

[4]All switches use the same function $\mathcal{R}(s, i)$ that takes as input a switch identifier s and a number i, and outputs a new identifier unique to the inputs. A physical switch w only advertises in link-state advertisements its own physical switch identifier s_w and the number L of virtual switches it is currently running. Every switch can then determine the virtual identifiers of w by computing $\mathcal{R}(s_w, i)$ for $1 \leq i \leq L$.

3.2 Simple and Reliable Handling of Switch Failures and Recoveries

The switch-level topology can change if a new switch/link is added to the network, an existing switch/link fails, or a previously failed switch/link recovers. These failures may or may not partition the network into multiple disconnected components. Link failures are typically more common than switch failures, and partitions are very rare if the network has sufficient redundancy.

Handling Events that Do Not Modify the Set of Live Switches. In the case of a link failure/recovery that does not partition a network, the set of switches appearing in the link-state map does not change. Since the hash function $\mathcal{F}$ is defined with the set of live switches in the network, a link failure/recovery does not modify $\mathcal{F}$; given a key, its resolver will not change. Hence, all that needs to be done is to update the link-state map to ensure packets continue to traverse new shortest paths. In SEATTLE, this is simply handled by the link-state protocol.

Handling Events that Modify the Set of Live Switches. However, if a switch fails or recovers, the set of live switches in the link-state map changes. Hence there may be some keys k whose old resolver r_k^{old} (i.e., k's resolver before the switch failure or recovery) differs from its new resolver r_k^{new}. To deal with this, the tuple (k, v) must be moved from r_k^{old} to r_k^{new}. This is handled by having the switch s_k that originally published (k, v) monitor the liveness of k's resolver through link-state advertisements—which can be done with no additional control overhead because the publisher switch s_k participates in the link-state protocol. When s_k detects that r_k^{new} differs from r_k^{old}, it republishes (k, v) to r_k^{new}. The value (k, v) is eventually removed from r_k^{old} after a timeout, ensuring availability of (k, v) during convergence. Additionally, when a value v denotes a location (e.g., a switch id s) and s goes down, each switch scans the list of locally-stored (k, v) pairs and removes all entries whose value v equals s. Note this procedure correctly handles network partitions because the link-state protocol ensures that each switch will be able to see only switches present in its partition.

3.3 Reducing Lookup Latency and Enabling Fault Isolation via Multilevel DHT

The SEATTLE design presented so far leverages the link-state routing protocol to partition a large number of frequently changing key-value pairs (specifically host information) among a distributed set of switches and to ensure eventual consistency of the key-value pairs. Running a single network-wide link-state routing protocol, however, might not be easily achievable in large networks that are susceptible to frequent topological changes (switch/link failures and recoveries) because each such topological change results in network-wide link-state advertisements. While flooding link-state advertisements itself can consume increasingly more resources as a network size grows, exchanging a large amount of host information to maintain the correctness of the DHT system after a topological change can consume even more resources and cause transient inconsistency in the directory system. Running a single routing protocol instance over a large set of switches may also be inappropriate if network operators wish to provide stronger fault isolation among geographic regions or to run heterogeneous routing protocols for administrative purposes. Moreover, when a SEATTLE network is deployed over a wide area, the resolver could lie physically far from both the source and destination. Resolution through a long-distance communication increases latency and lowers reliability.

To deal with these problems, SEATTLE may be configured hierarchically by leveraging a multilevel, one-hop DHT illustrated in Figure 2. This hierarchical configuration

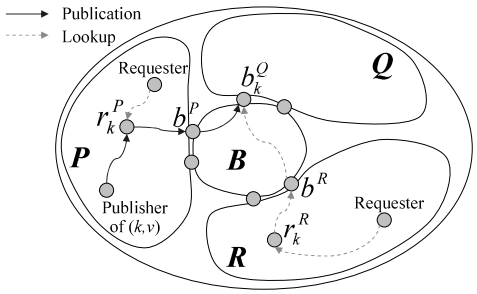


Fig. 2. Hierarchical SEATTLE hashes keys onto *regions*.

of the DHT system, however, does not require end hosts to adopt hierarchical addressing; the multilevel DHT system still ensures flat addressing.

3.3.1 Dividing the Network into Small Regions. The core idea of our multilevel one-hop DHT design is divide-and-conquer. Specifically, we divide a large SEATTLE network into several smaller *regions* (regions P, Q, and R in Figure 2) and a *backbone* region (B in Figure 2) interconnecting the other regions. Each region is composed of a subset of switches in the network that are located close to one another and the hosts attached to those switches. A region is then connected to the backbone via its *border* switch(es) (e.g., b^P connecting region P to B) which advertises its special role to all other switches in the region via the link-state protocol. The backbone is composed of the border switches of all regions (b^P, b^Q, and b^R). The switch-level topology information about a region is then summarized in a manner similar to the OSPF *areas*, keeping most link-state advertisements only within a small region. This allows each switch in a region to know about all the other switches in the same region only (including the region's border switch) and avoid learning unnecessary information about switches in different regions.

Our multilevel one-hop DHT aims to ensure two key properties. First, all regional host-information lookups are handled strictly within their own regions; only inter-region lookups are forwarded across the backbone, substantially reducing overall lookup latency in a large network. Second, a network failure or recovery event in a region is completely hidden from switches in other regions, hence eliminating unnecessary network-wide link-state flooding and avoiding host-information re-partitioning in other regions.

3.3.2 Running a Separate, Self-Contained Hash Ring in Each Region. The specific mechanism SEATTLE employs to achieve these goals is separating *regional* hash rings (DHT systems) from the backbone hash ring, and using the backbone ring only for inter-regional lookups. A regional ring is populated with all switches in the region, and the backbone ring is populated with the border switches from all regions. Suppose a host k's information v originates from a switch in region P. The (k, v) pair is first published to the regional resolver r_k^P, which is k's resolver in region P dictated by the regional ring. In addition to locally storing (k, v), r_k^P also forwards (k, v) to one or more border switches of the region (e.g., b^P). The role of this border switch b^P is simple; on behalf of the original publisher of (k, v) in region P, b^P acts as a proxy publisher of (k, v) in the backbone. That is, b^P stores (k, v) locally, hashes k onto the backbone ring, and publishes (k, v) to another backbone switch b_k^Q. Note that b_k^Q is the backbone resolver for k determined by the backbone ring, which is also a border switch of region Q. Finally, b_k^Q stores (k, v) for future inter-regional lookups.

Any local lookup on k arising in region P is then resolved by r_k^P. On the other hand, when a nonlocal lookup is performed, for example, if a switch in region R wishes to lookup k, the lookup is first forwarded to the local resolver for r_k^R in region R. Since r_k^R does not maintain any value associated with k, it forwards the lookup to the border switch b^R and triggers an inter-regional lookup. b^R then forwards the lookup to b_k^Q as dictated by the backbone ring.

3.3.3 Further Homogenizing Workloads on Switches. Note this mechanism imposes a larger workload on backbone (i.e., border) switches than on nonbackbone switches because each backbone switch maintains all key-value pairs originating from its own region, as well as some other key-value pairs published to itself for inter-regional lookup. In essence, the backbone switches collectively maintain all hosts' information in the entire network, whereas switches in a nonbackbone region maintain only the information about the hosts in that particular region. While it is fair to assume that administrators, in practice, would use more powerful (with a larger forwarding table and faster CPUs, for example) switches to build a backbone, it may also be desirable to eliminate this constraint for some networks.

As an optimization to reduce load on backbone switches, b^P may *relay* (k, v) rather than store it itself. In addition, k's backbone resolver b_k^Q may also avoid storing (k, v) itself by hashing k against its own regional ring for Q and storing (k, v) at a switch in Q. Doing so, however, introduces some cost. For example, when b_k^Q fails, the border switch b^P of region P cannot easily re-publish (k, v) to a new resolver in the backbone because b^P does not keep (k, v) in its local forwarding table. In general, because switch failures and recoveries (i.e., link-state updates) are not propagated across regions, it becomes increasingly difficult to maintain the correctness of the directory service by re-publishing hosts information in a timely fashion. As a solution, each original publisher switch periodically sends out publication messages to update the (k, v) pairs in the resolvers that lie outside of its region. To improve availability even further, a border switch can also replicate (k, v) at multiple backbone resolvers (as described in Section 3.1.2) and issue multiple simultaneous lookups in parallel.

4. SCALING ETHERNET WITH THE NETWORK-LAYER ONE-HOP DHT

The previous section described the design of a distributed network-layer directory service based on a one-hop DHT. In this section, we describe how we specifically use the directory service to enable scalable address resolution and efficient packet delivery. We first briefly describe how to forward data packets to MAC addresses in Section 4.1. We then describe our remaining contributions: an optimization that eliminates the need to explicitly look up host location in the DHT by piggy-backing that information on ARP requests in Section 4.2, and a scalable, dynamic cache-update protocol in Section 4.3.

4.1 Host Location Resolution

Hosts use the directory service described in Section 3 to publish and maintain mappings between their MAC addresses and their current locations. These mappings are used to forward data packets, using the procedure shown in Figure 3. When a host a with MAC address mac_a first arrives at its access switch s_a, the switch must publish a's MAC-to-location mapping in the directory service. Switch s_a does this by computing $\mathcal{F}(mac_a) = r_a$, and instructing r_a to store (mac_a, s_a). We refer to r_a as the *location resolver* for a. Then, if some host b connected to switch s_b wants to send a data packet to mac_a, b forwards the data packet to s_b, which in turn computes $\mathcal{F}(mac_a) = r_a$. Switch

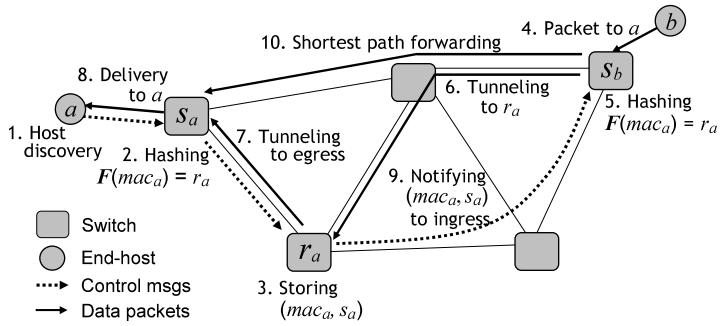


Fig. 3. Packet forwarding and lookup in SEATTLE.

s_b then forwards the packet to r_a. Since r_a may be several hops away, s_b encapsulates the packet with an outer header with r_a's address as the destination. Switch r_a then looks up a's location s_a, and forwards the packet towards s_a again using encapsulation.

Meanwhile, in order to limit the number of data packets traversing the resolver, r_a also informs s_b of a's current location s_a, which switch s_b then caches for later use. While forwarding the first few packets of a flow via a resolver switch increases path lengths and can potentially impact TCP performance because of the path change, we show in Section 8 that this causes only negligible impact to the application performance and explain why. More importantly, in the next section we also introduce an optimization that allows data packets to traverse only shortest paths by piggy-backing location information on ARP replies.

SEATTLE uses encapsulation to deliver packets between an ingress and a resolver switch, a resolver and an egress switch, or between an ingress and an egress switch. This mechanism does introduce a small additional bandwidth overhead (e.g., 14 bytes when using the EtherIP encapsulation [Housley and Hollenbeck 2002]) to every packet. Administrators can easily reduce this overhead by employing jumbo frames, which are widely available in today's Ethernet bridges and IP routers.

Note SEATTLE manages per-host information via reactive resolution, as opposed to the proactive dissemination scheme used in previous approaches [Myers et al. 2004; Perlman 2004; Rodeheffer et al. 2000]. The scaling benefits of this reactive resolution increase in enterprise, data-center, and access-provider networks because most hosts in such networks communicate with a small number of popular hosts, such as mail/file/Web servers, traffic loadbalancers, printers, VoIP gateways, and Internet gateways [Aiello et al. 2005]. To prevent forwarding tables from growing unnecessarily large, the access switches can also apply various cache-management policies. For correctness, however, the cache-management scheme must not evict the information of those hosts that are directly connected to the switch or are registered with the switch for resolution. Unlike Ethernet bridging, cache misses in SEATTLE do not lead to flooding, making the network resistant to cache poisoning attacks (e.g., forwarding-table overflow attack) or a sudden shift in traffic patterns. Moreover, those switches that are not directly connected to end hosts (i.e., aggregation or core switches) do not need to maintain any cached entries.

4.2 Host Address Resolution

In conventional Ethernet, a host sending an IP packet first broadcasts an ARP request to look up the MAC address of the host owning the destination IP address contained in the request. To enhance scalability, SEATTLE avoids this broadcast-based ARP

operation. In addition, we extend ARP to return both the location and the MAC address of the end host to the requesting switch. This allows data packets following an ARP query to directly traverse shortest paths.

Broadcast-Free ARP. SEATTLE replaces the traditional broadcast-based ARP with an extension to the one-hop DHT directory service. In particular, switches use $\mathcal{F}$ with an IP address as the key. Specifically, when host a arrives at access switch s_a, the switch learns a's IP address ip_a (using techniques described in Section 5.1), and computes $\mathcal{F}(ip_a) = v_a$. The result of this computation is the identifier of another switch v_a. Finally, s_a informs v_a of (ip_a, mac_a). Switch v_a, the address resolver for host a, then uses the tuple to handle future ARP requests for ip_a redirected by other remote switches. Note that host a's location resolver (i.e., $\mathcal{F}(mac_a)$) may differ from a's address resolver (i.e., $\mathcal{F}(ip_a)$).

Optimizing Forwarding Paths via ARP. For hosts that issue an ARP request, SEATTLE eliminates the need to perform forwarding via the location resolver as mentioned in Section 4.1. This is done by having the address resolver switch v_a also maintain the location of a (i.e., s_a) in addition to mac_a. Upon receiving an ARP request from some host b, the address resolver v_a returns both mac_a and s_a back to b's access switch s_b. Switch s_b then caches s_a for future packet delivery, and returns mac_a to host b. Any packets sent by b to a are then sent directly along the shortest path to a.

It is, however, possible that host b already has mac_a in its ARP cache and immediately sends data frames destined to mac_a without issuing an ARP request in advance. Even in such a case, as long as s_b also maintains a's location associated with mac_a, s_b can forward those frames correctly. To ensure access switches cache the same entries as hosts, the timeout value that an access switch applies to the cached location information should be larger than the ARP cache timeout used by end hosts.[5] Note that, even if the contents of the switch cache gets different from that of the host cache (due to switch reboot, etc.), SEATTLE continues to operate correctly because switches can resolve a host's location by hashing the host's MAC address to the host's location resolver.

4.3 Handling Host Dynamics with Low Control-Plane Overhead

End hosts can undergo three kinds of changes in a SEATTLE network. First, a host may change location, for example if it has physically moved to a new location (e.g., wireless hand-off), if its link has been plugged into a different access switch, or if it is a virtual machine and has migrated onto a new physical machine that allows the VM to retain its MAC address. Second, a host may change its MAC address, for example, if its network-interface card (NIC) is replaced, if it is a VM and has migrated onto a new physical machine that requires the VM to use the physical machine's MAC address, or if multiple hosts collectively acting as a single server or router (to ensure high availability) experience a fail-over event [Hinden 2004]. Third, a host may change its IP address, for example if a DHCP lease expires, or if the host's address is manually reconfigured. In practice, multiple of these changes may occur simultaneously. To ensure correct packet delivery when these changes occur, we need to keep the directory service up-to-date.

SEATTLE handles these changes by modifying the contents of the directory service via *insert*, *delete*, and *update* operations. An insert operation adds a new (k, v) pair to the DHT, a delete operation removes a (k, v) pair from the DHT, and the update operation changes the value v associated with k. First, in the case of a location change,

[5]The default setting of the ARP cache timeout in most common operating systems ranges 5 to 20 minutes.

the host h moves from one access switch s_h^{old} to another s_h^{new}. In this case, s_h^{new} inserts a new MAC-to-location entry. Since h's MAC address already exists in the DHT, this action will update h's old location with its new location. Second, in the case of a MAC address change, h's access switch s_h inserts an IP-to-MAC entry containing h's new MAC address, causing h's old IP-to-MAC mapping to be updated. Since a MAC address is also used as a key of a MAC-to-location mapping, s_h deletes h's old MAC-to-location mapping and inserts a new mapping, respectively with the old and new MAC addresses as keys. Third, in the case of an IP address change, we need to ensure that future ARP requests for h's old IP address are no longer resolved to h's MAC address. To ensure this, s_h deletes h's old IP-to-MAC mapping and insert the new one. Finally, if multiple changes happen at once, the above steps occur simultaneously.

Ensuring Seamless Mobility. As an example, consider the case of a mobile host h moving between two access switches, s_h^{old} and s_h^{new}. To handle this, we need to update h's MAC-to-location mapping to point to its new location. As described in Section 4.1, s_h^{new} inserts (mac_h, s_h^{new}) into r_h upon arrival of h. Note that the location resolver r_h selected by $\mathcal{F}(mac_h)$ does *not* change when h's location changes. Meanwhile, s_h^{old} deletes (mac_h, s_h^{old}) when it detects h is unreachable (either via timeout or active polling). Additionally, to enable prompt removal of stale information, the location resolver r_h informs s_h^{old} that (mac_h, s_h^{old}) is obsoleted by (mac_h, s_h^{new}).

However, host locations cached at other access switches (for shortest-path forwarding) must be kept up-to-date as hosts move. SEATTLE solves this problem by introducing a *reactive cache-update protocol* which takes advantage of the fact that, even after updating the information at r_h, s_h^{old} may receive packets destined to h because other access switches in the network might have the stale information in their forwarding tables. Hence, when s_h^{old} receives packets destined to h, it explicitly notifies ingress switches that sent the mis-delivered packets of h's new location s_h^{new}. To minimize service disruption, s_h^{old} also forwards those misdelivered packets s_h^{new}.

Unfortunately, these efforts to keep a host's location up-to-date can be wasted, if the host changes its location too frequently. To avoid such a waste, a resolver switch can selectively disable ingress notification for a highly mobile host – for example, by monitoring the host's mobility rate – and thus devolve to acting as a relay for packets destined to the host. This mechanism essentially trades latency (i.e., path stretch) for efficiency and consistency in replicating the mobile host's information. Since a mobile host is likely to experience poor communication quality (e.g., higher loss rate or lower capacity at the physical layer) anyway, this trade-off would not over-penalize the mobile host.

Updating Remote Hosts' ARP Tables. In addition to updating contents of the directory service, some host changes require informing other hosts in the system about the change. For example, if a host h changes its MAC address, other hosts who happened to store h's old MAC address mac_h^{old} in their local ARP caches must be able to update the obsolete MAC address immediately. In conventional Ethernet, this is achieved by broadcasting a *gratuitous ARP request* originated by h [Gratuitous ARP 2009]. A gratuitous ARP is an ARP request containing the sending host's MAC and IP addresses. This request is not a query for a reply, but is instead a notification to update other end hosts' ARP tables and to detect IP address conflicts on the subnet. Relying on broadcast to update other hosts clearly does not scale to large networks. SEATTLE avoids this problem by unicasting gratuitous ARP packets only to hosts with invalid mappings. This is done by having s_h maintain a *MAC revocation list*.

Upon detecting h's MAC address change, switch s_h inserts $(ip_h, mac_h^{old}, mac_h^{new})$ in its revocation list. From then on, whenever s_h receives a packet whose destination

(IP, MAC) address pair equals (ip_h, mac_h^{old}), it sends a unicast gratuitous ARP request containing (ip_h, mac_h^{new}) to the source host which sent those packets. Note that, when both h's MAC address and location change at the same time, the revocation information is created at h's old access switch by h's address resolver $v_h = \mathcal{F}(ip_h)$. To minimize service disruption, s_h also informs the source host's ingress switch of (mac_h^{new}, s_h) so that the packets destined to mac_h^{new} can then be directly delivered to s_h, avoiding an additional location lookup.

Note this approach of updating remote ARP caches does not require s_h to look up every packet's IP and MAC address pair against the revocation list because s_h can skip the lookup in the common case—a switch's revocation list is likely to remain empty most time because, unlike location changes, hosts' IP-to-MAC address mappings change rarely. In typical enterprise or data-center networks, such change happens only when a host gets a new network-interface card, or a pair of hosts sharing the same IP address (typically an active/stand-by setup) swaps their active state [Hinden 2004]. Entries in the revocation list are removed after a timeout set equal to the ARP cache timeout of end hosts.

5. ENSURING BACKWARDS-COMPATIBILITY WITH ETHERNET

To be fully backwards-compatible with conventional Ethernet, SEATTLE must act like conventional Ethernet from the perspective of end hosts. This goal translates into a few specific requirements. First, the way hosts interact with the network to *bootstrap* themselves (e.g., to acquire their own or other hosts' addresses, and to expose their presence to switches) must be the same as Ethernet. Second, switches have to support general broadcast traffic which uses broadcast/multicast Ethernet addresses as destinations. Third, to scope broadcast traffic and to control reachability among hosts, administrators should be able to group end hosts into VLANs. In this section, we describe how SEATTLE satisfies these requirements without incurring the scalability challenges of traditional Ethernet.

5.1 Bootstrapping Hosts

Host Discovery by Access Switches. When an end host arrives at a SEATTLE network, its access switch needs to discover the host's MAC and IP addresses. To discover a new host's MAC address, SEATTLE switches use the same MAC learning mechanism as conventional Ethernet on all host-facing ports. On the other hand, to learn a new host's IP address or detect an existing host's IP address change, SEATTLE switches snoop on gratuitous ARP requests — most operating systems today generate a gratuitous ARP request when the host boots up, the host's network interface or links comes up, or an address assigned to the interface changes [Gratuitous ARP 2009]. Even if a host does not generate a gratuitous ARP, the switch can still learn the host's IP address via snooping on DHCP messages. Similarly, when an end host fails or disconnects from the network, the access switch is responsible for detecting that the host has left, and deleting the host's information from the network.

Host Configuration without Broadcasting. For scalability, SEATTLE resolves DHCP messages without broadcasting. When an access switch receives a broadcast DHCP discovery message from an end host, the switch delivers the message directly to a DHCP server via unicast, instead of broadcasting it. SEATTLE implements this mechanism using the existing DHCP relay agent standard [Droms 1997]. This standard is used when an end host needs to communicate with a DHCP server outside the host's broadcast domain. The standard proposes that a host's IP gateway forward a DHCP discovery to a DHCP server via IP routing. In SEATTLE, a host's access

switch can perform the same function with Ethernet encapsulation. Access switches can discover a DHCP server using a similar approach to the service discovery mechanism introduced in Section 3.1.2. For example, the DHCP server hashes the string "DHCP_SERVER" to a switch, and then stores its location at that switch. Other switches then forward DHCP requests using the hash of the string.

5.2 Groups: Scalable and Flexible VLANs

SEATTLE avoids unnecessary broadcasting by eliminating flooding of unicast packets and supporting ARP and DHCP via unicast-based resolution. However, to offer the same semantics as Ethernet bridging, SEATTLE still needs to support general broadcasting—transmission of packets sent to a broadcast or multicast MAC addresses. In fact, some popular applications (e.g., IP multicast applications, peer-to-peer file sharing programs) frequently use subnet-wide broadcasting for service-discovery or contents-distribution purposes. In a network built with a large number of switches and end hosts, however, performing network-wide broadcasts significantly overloads both end hosts and switches, largely wasting data-plane resources. To cope with this problem, SEATTLE needs to offer a broadcast scoping mechanism similar to VLANs. VLANs in Ethernet, however, are used not only for broadcast scoping, but also for controlling reachability between hosts. Therefore, the SEATTLE mechanism that replaces VLAN should also enable *access control*.

To offer the VLAN semantics to end hosts, SEATTLE introduces a notion of *group*. Similar to a VLAN, a group is defined as a set of hosts forming a shared broadcast domain irrespective of their locations. Hence, broadcast traffic from a host is delivered only to the hosts in the same group, enabling broadcast scoping. Unlike VLANs, however, SEATTLE groups do *not* necessarily deny reachability between hosts in different groups because SEATTLE switches can resolve any host's address and location without relying on broadcasting. Thus, SEATTLE groups provide several additional benefits over VLANs. First, in contrast to VLANs, groups do not need to be manually trunked along the paths between switches.[6] Rather, a group is automatically extended to cover any switch as soon as a member of that group arrives at the switch. Second, a group is not forced to correspond to a single IP subnet and hence may span multiple subnets or a portion of a subnet, if desired. Third, unicast reachability in layer-2 between two different groups may be flexibly determined depending on access-control policies—a rule set defining which groups can communicate with which—between the groups.

This flexibility of SEATTLE groups allows administrators to conveniently implement various useful network designs that were hard to achieve with VLANs. To show the benefits of this approach, we give three motivating examples. First, by aligning a group with a subnet and by denying direct reachability between groups, one can simply implement exactly the same functionality as VLANs. However, groups can contain a much larger number of end hosts than a VLAN and can be extended to anywhere in the network without harming control-plane scalability and data-plane efficiency. Second, by defining a group as a fraction of an IP subnet and by prohibiting inter-group reachability, one can ensure the same semantics as private VLAN (PVLAN) [Hucaby and McQuerry 2002], which is frequently used in hotel networks to avoid creating numerous tiny IP subnets. Unlike PVLANs, however, groups can be extended over

[6]While administrators of a SEATTLE network still need to associate each host-facing port with its group, several well-known principles and solutions exist for this problem and are readily available today. For example, VLAN-management systems that can automate this task (e.g., mapping a MAC address, a physical port, a cryptographic identity, or even a traffic flow to a VLAN) are already available and deployed in many networks [Tengi et al. 2004]. SEATTLE can employ the same solutions.

multiple bridges and scale to a large network. Finally, by permitting direct layer-2 reachability between groups, one can both scope broadcast traffic in each VLAN and optimize traffic forwarding paths between hosts in different groups, avoiding latency and vulnerability due to forwarding through IP gateways.

Group-Wide Broadcasting via Multicast Trees. VLAN trunking in conventional Ethernet is essentially equivalent to the process of manually configuring routes for each VLAN along which unicast and broadcast traffic is delivered. To avoid such an operational burden, SEATTLE supports general broadcasting by having all broadcast packets within a group be delivered through a multicast tree sourced at a dedicated switch, namely a *broadcast root*, specific to the group. The mapping between a group and its broadcast root is determined by hashing the group's identifier to a switch via $\mathcal{F}$ – the same consistent-hash function used for any other mapping tasks. Switches construct a multicast tree in a manner similar to IP multicast (using link-state information) and thus inherit the safety (i.e., loop freedom) and efficiency (i.e., delivering broadcast traffic only when necessary) of IP multicast. The specific mechanism is as follows. When a switch first detects an end host that is a member of group g, the switch issues a join message that is carried up to the nearest graft point on the multicast tree toward g's broadcast root. When the host sends a broadcast packet, its access switch marks the packet with g's id and forwards it along g's multicast tree. Finally, when the host leaves g, its access switch may prune the branch.

Group-Based Access Control. In addition to handling broadcast traffic, groups in SEATTLE also provide a highly scalable and flexible namespace upon which inter-host reachability policies are defined. Note that groups only offer the namespace for access control but do not limit reachability themselves. The specific mechanism is as follows. When a host a arrives at its access switch s_a, a's group membership is determined by s_a and published to a's resolver r_a along with a's other information. Access control policies are then enforced by r_a at a lookup time when another host b attempts to resolve a's information. In particular, r_a replies to b's lookup request only if the access-control policy between the a's group and b's group permits reachability. When a's information is cached at another ingress switch, the ingress should also perform the same isolation mechanism as the resolver.

6. EVALUATING MACRO-SCALE PERFORMANCE VIA SIMULATIONS

To evaluate SEATTLE, we take a two-pronged approach. First, we use simulations to understand SEATTLE's scalability, efficiency and reliability under a wide variety of operating conditions. Using simulations enables us to examine SEATTLE's performance in large, dynamic networks in a reproducible environment. Second, to evaluate SEATTLE's micro-scale behavior and performance (e.g., packet processing latency, impact of failures on traffic flows and applications), we conduct an implementation of SEATTLE, along with a deployment on a testbed and evaluation with application traffic.

In this section, we start by taking the former approach, while deferring the latter to Section 8. We start this section by describing our simulation environment. Then, we evaluate SEATTLE's performance under workloads collected from several real operational networks. Finally we investigate SEATTLE's performance in dynamic environments by generating host mobility and topology changes.

6.1 Methodology for Large-Scale, Packet-Level Simulations

To evaluate the performance of SEATTLE, we would ideally like to have several pieces of information, including complete layer-two topologies from a number of

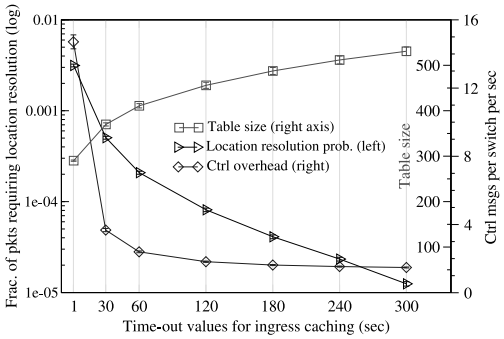
representative enterprises and access providers, traces of all traffic sent on every link in their topologies, the set of hosts at each switch/router in the topology, and a trace of host movement patterns. Unfortunately, administrators or large production networks (understandably) were not able to share this detailed information with us due to privacy concerns, and also because they typically do not log events on such large scales. Hence, we leveraged real traces where possible, and supplemented them with synthetic traces. To generate the synthetic traces, we made realistic assumptions about workload characteristics, and varied these characteristics to measure the sensitivity of SEATTLE to our assumptions.

In our packet-level simulator, we replayed packet traces collected from the Lawrence Berkeley National Lab campus network [Pang et al. 2005]. There are four sets of traces, each collected over a period of 10 to 60 minutes, containing traffic to and from roughly 9,000 end hosts distributed over 22 different subnets. The end hosts were running various operating systems and applications, including malware (some of which engaged in scanning). To evaluate sensitivity of SEATTLE to network size, we artificially injected additional hosts into the trace. We did this by creating a set of virtual hosts, which communicated with a set of random destinations, while preserving the distribution of destination-level popularity of the original traces. We also tried injecting MAC scanning attacks and artificially increasing the rate at which hosts send [Allman et al. 2007].

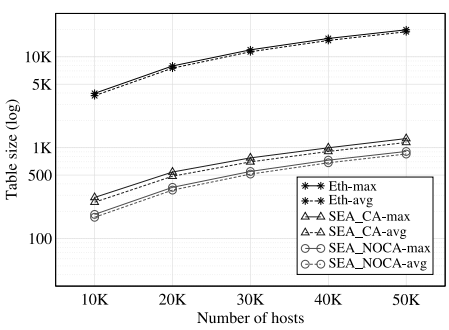
We measured SEATTLE's performance on four representative topologies. *Campus* is the campus network of a large (roughly 40,000 students) university in the United States, containing 517 routers and switches. *AP-small* (AS 3967) is a small access provider network consisting of 87 routers, and *AP-large* (AS 1239) is a larger network with 315 routers [Spring et al. 2004]. Because SEATTLE switches are intended to replace both IP routers and Ethernet bridges, the routers in these topologies are considered SEATTLE switches in our evaluation. To investigate a wider range of environments, we also constructed a model topology called *DC*, which represents a typical data center network composed of four full-meshed core routers each of which is connected to a mesh of twenty one aggregation switches. This roughly characterizes a commonly-used topology in data centers [Arregoces and Portolani 2003].

Our topology traces were anonymized, and hence lack information about how many hosts are connected to each switch. To deal with this, we leveraged CAIDA Skitter traces [Skitter 2010] to roughly characterize this number for networks reachable from the Internet. However, since the CAIDA Skitter traces form a sample representative of the wide-area networks, it is not clear whether they apply to the smaller-scale networks we model. Hence, for *DC* and *Campus*, we assume that hosts are evenly distributed across leaf-level switches.

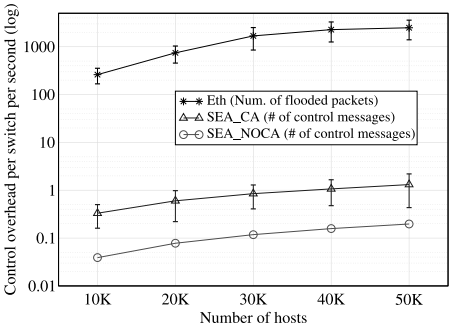
Given a fixed topology, the performance of SEATTLE and Ethernet bridging can vary depending on traffic patterns. To quantify this variation we repeated each simulation run 25 times, and plot the average of these runs with 99% confidence intervals. For each run we vary a random seed, causing the number of hosts per switch, and the mapping between hosts and switches to change. Additionally for the cases of Ethernet bridging, we varied spanning trees by randomly selecting one of the core switches as a root bridge. Our simulations assume that all switches are part of the same broadcast domain. However, since our traffic traces are captured in each of the 22 different subnets (i.e., broadcast domains), the traffic patterns among the hosts preserve the broadcast domain boundaries. Thus, our simulation network is equivalent to a VLAN-based network where a VLAN corresponds to an IP subnet, and all non-leaf Ethernet bridges are trunked with all VLANs to enhance mobility.



(a) Sensitivity to cache-eviction timeout



(b) Forwarding table size



(c) Control overhead

Fig. 4. (a) Effect of cache timeout in *AP-large* with 50*K* hosts, (b) Table size increase in *DC*, and (c) Control overhead in *AP-large*. Error bars in these figures show confidence intervals for corresponding data points. A sufficient number of simulation runs reduced these intervals.

6.2 Control-Plane Scalability

Sensitivity to Cache Eviction Timeout. SEATTLE caches host information to route packets via shortest paths and to eliminate redundant resolutions. If a switch removes a host-information entry before a locally attached host does (from its ARP cache), the switch will need to perform a location lookup to forward data packets sent by the host. To eliminate the need to queue data packets at the ingress switch, those packets are forwarded through a location resolver, leading to a longer path. To evaluate this effect, we simulated a forwarding table management policy for switches that evicts unused entries after a timeout. Figure 4(a) shows performance of this strategy across different timeout values in the *AP-large* network. First, the fraction of packets that require data-driven location lookups (i.e., lookups not piggy-backed on ARPs) is very low and decreases quickly with larger timeout. Even for a very small timeout value of 60 seconds, over 99.98% of packets are forwarded without a separate lookup. We also confirmed that the number of data packets forwarded via location resolvers drops to zero when using timeout values larger than 600 seconds (i.e., roughly equal to the ARP cache timeout at end hosts). Also control overhead to maintain the directory decreases quickly, whereas the amount of state at each switch increases moderately with larger timeout. Hence, in a network with properly configured hosts and reasonably small (e.g., less than 2% of the total number of hosts in this topology) forwarding tables, SEATTLE always offers shortest paths.

Forwarding Table Size. Figure 4(b) shows the amount of state per switch in the *DC* topology. To quantify the cost of ingress caching, we show SEATTLE's table size with and without caching (*SEA_CA* and *SEA_NOCA*, respectively). Ethernet requires more state than SEATTLE without caching, because Ethernet stores active hosts' information entries at almost every bridge. In a network with s switches and h hosts, each Ethernet bridge must be provisioned to store an entry for each destination, resulting in $O(sh)$ state requirements across the network. SEATTLE requires only $O(h)$ state since only the access and resolver switches need to store location information for each host. In this particular topology, SEATTLE reduces forwarding-table size by roughly a factor of 22. Although not shown here due to space constraints, we find that these gains increase to a factor of 64 in *AP-large* because there are a larger number of switches in that topology. While the use of caching significantly reduces the number of redundant location resolutions, we can see that it increases SEATTLE's forwarding-table size by roughly a factor of 1.5. However, even with this penalty, SEATTLE reduces table size compared with Ethernet by roughly a factor of 16. This value increases to a factor of 41 in *AP-large*. If desired, network administrators can vary the amount of space allocated to cache host entries, so as to trade off stretch and space requirements.

Control Overhead. Figure 4(c) shows the amount of control overhead generated by SEATTLE and Ethernet. We computed this value by dividing the total number of control messages – flooded packets in case of Ethernet, or host-information exchange messages in case of SEATTLE – over all links in the topology by the number of switches, then dividing by the duration of the trace. SEATTLE significantly reduces control overhead as compared to Ethernet. This happens because Ethernet generates network-wide floods for a significant number of packets, while SEATTLE leverages unicast to disseminate host location. Here we again observe that use of caching degrades performance slightly. Specifically, the use of caching (*SEA_CA*) increases control overhead roughly from 0.1 to 1 packet per second as compared to *SEA_NOCA* in a network containing 30K hosts. However, *SEA_CA*'s overhead still remains a factor of roughly 1000 less than in Ethernet, and this ratio corresponds roughly to the number of links in the topology. In general, we found that the difference in control overhead increased roughly with the number of links in the network.

Comparison with ID-Based Routing Approaches. We implemented the ROFL, UIP, and VRR protocols in our simulator. To ensure a fair comparison, we used a link-state protocol to construct vset-paths [Caesar et al. 2006a] along shortest paths in UIP and VRR, and created a UIP/VRR node at a switch for each end host the switch is attached to. Performance of UIP and VRR was quite similar to performance of ROFL with an unbounded cache size. Figure 5(a) shows the average relative latency penalty, or *stretch*, of SEATTLE and ROFL in the *AP-large* topology. We measured stretch by dividing the time the packet was in transit by the delay along the shortest path through the topology. Overall, SEATTLE incurs smaller stretch than ROFL. With a cache size of 1000, SEATTLE offers a stretch of roughly 1.07, as opposed to ROFL's 4.9. This happens because (i) when a cache miss occurs, SEATTLE resolves location via a single-hop rather than a multi-hop lookup, and (ii) SEATTLE's caching is driven by traffic patterns, and hosts in an enterprise network typically communicate with only a small number of popular hosts. Note that SEATTLE's stretch remains below 5 even when a cache size is 0. Hence, even with worst-case traffic patterns (e.g., when every host communicates with all other hosts, or when switches keep very small caches), SEATTLE still ensures reasonably small stretch. Finally, we compare path stability with ROFL in Figure 5(b). We vary the rate at which hosts leave and join the network, and measure path stability as the number of times a flow changes its path (the sequence of switches it traverses)

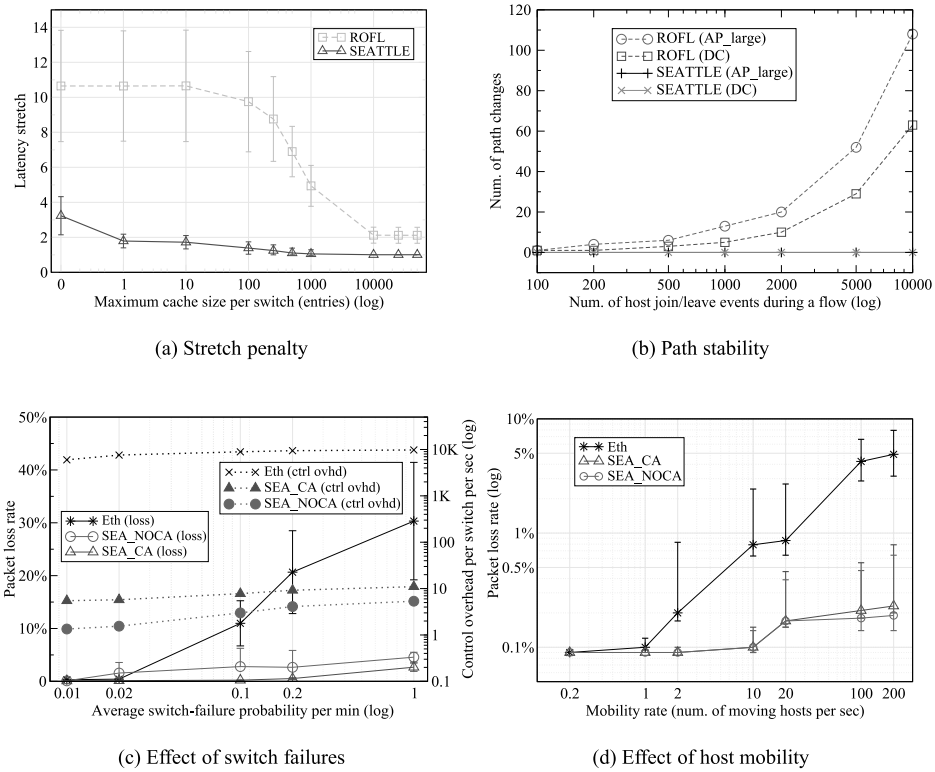


Fig. 5. (a) Stretch across different cache sizes in *AP-large* with 10*K* hosts, (b) Path stability, (c) Effect of switch failures in *DC*, and (d) Effect of host mobility in *Campus*.

in the presence of host churn. We find that ROFL has over three orders of magnitude more path changes than SEATTLE.

6.3 Sensitivity to Network Dynamics

Effect of Network Changes. Figure 5(c) shows performance during switch failures and recoveries. Here, we cause switches to fail at rates drawn from a Pareto distribution (α = 2.0) with varying mean values (0.01 to 1 failure per minute). Switch recovery times are also drawn from the Pareto distribution with a fixed mean of 30 seconds. Some of this experiment setup represents a very harsh operational condition; having each switch fail once every minute on average with a mean-time-to-recovery of 30 seconds causes one-third of all switches in the network to remain failed at any given time.

Across all failure rates we found SEATTLE is able to deliver a larger fraction of packets than Ethernet. This happens because SEATTLE is able to use all links in the topology to forward packets, while Ethernet can only forward over a spanning tree. Additionally, after a switch failure, Ethernet must recompute this tree, which causes outages until the process completes. Although forwarding traffic through a location resolver in SEATTLE causes a flow's fate to be shared with a larger number of switches, we found that availability remained higher than that of Ethernet. Additionally, ingress caching improved availability further because hosts' location and address information in an ingress cache remain valid as long as the hosts still reside in the network. Most

notably, the number of control messages SEATTLE switches exchange to adapt to the new topology is fewer by nearly three orders of magnitude than that of the flooded messages Ethernet bridges exchange until they re-learn hosts' location.

Effect of Host Mobility. To investigate the effect of physical or virtual host mobility on SEATTLE performance, we randomly move hosts between access switches. We drew mobility times from a Pareto distribution with $\alpha = 2.0$ and varying means. For high mobility rates, SEATTLE's loss rate is lower than Ethernet (Figure 5(d)). This happens because when a host moves in Ethernet, it takes some time for switches to evict stale location information, and learn the host's new location. Although some host operating systems broadcast a gratuitous ARP when a host moves, this increases broadcast overhead. In contrast, SEATTLE provides both low loss and low broadcast overhead by updating host state via unicasts.

7. IMPLEMENTATION USING OPEN-SOURCE NETWORKING SOFTWARE

Our simulation results shown in Section 6 indicate that SEATTLE performs efficiently on several network topologies. To verify SEATTLE's performance and practicality through a real deployment, we also implemented a prototype SEATTLE switch using two open-source routing software platforms: Click [Kohler et al. 2000] and XORP [Handley et al. 2005]. Figure 6 shows the overall structure of our implementation.

SEATTLE's control plane is divided into two functional components: (i) the module maintaining the switch-level topology, and (ii) the module maintaining and exchanging end-host information, maintaining a consistent hash containing all live switches' identifiers, and keeping forwarding information needed to choose specific next hops for packets. We use XORP to realize the first functional module, and extend Click to implement the second. Finally, we also extend Click to implement SEATTLE's data-plane—namely SEATTLE Forwarding Engine or SFE in Figure 6—which uses the host and switch information available in the control-plane and performs per-packet operations, such as host-information lookup, encapsulation, decapsulation, and next-hop forwarding. We explain the upper half of the SEATTLE control plane implemented in XORP in Section 7.1 and the lower half in Section 7.2. Then we describe the SEATTLE data-plane details in Section 7.3. Our control- and data-plane modifications to Click are specifically implemented as several Click elements shown in Figure 6. The entire box named Click in Figure 6 can be either instantiated as a user process or a Linux kernel-thread.

7.1 Maintaining the Switch-Level Topology with XORP

XORP is an extensible open-source software router [Handley et al. 2005], which consists of several routing protocols and management functions. As previously mentioned, SEATTLE relies on a link-state protocol to provide reachability between switches. We used XORP's OSPF protocol daemon to provide this function. In particular, each SEATTLE switch runs a XORP OSPF process and build a complete switch-level network map by exchanging link-state advertisements (LSAs) with other switches. Based on this network map, the XORP Routing Information Base Daemon (RIBD) runs Dijkstra's shortest-path algorithm and construct a routing table for the switch. Whenever this routing table is created or modified, RIBD installs the table into the data-plane module, which we implement with either a user-level Click process or a kernel-level Click thread. The data-plane module (i.e., the Click process instance in the user or kernel space) uses this finalized version of routing table, namely NextHopTable, to determine a next-hop switch for a destination switch. The Forwarding Engine Abstraction (FEA)

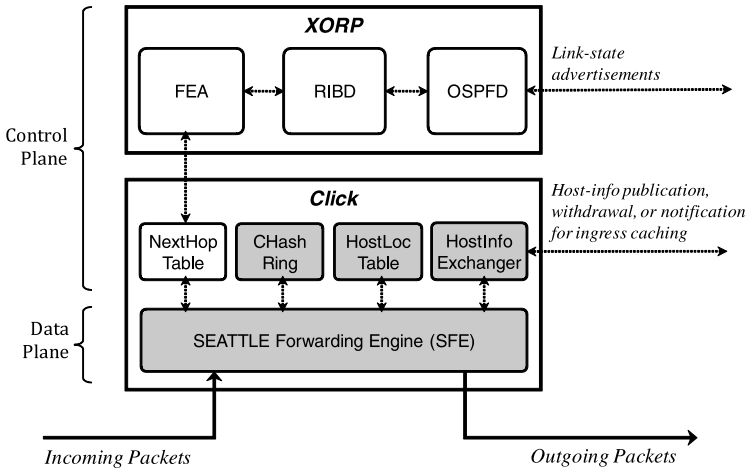


Fig. 6. Abstract diagram of our SEATTLE switch implementation. Unshaded boxes represent existing modules in XORP and Click that SEATTLE directly employs without any modification, whereas shaded boxes represent components that are written from scratch for this prototype. Dotted lines between the modules represent control-message exchanges, and the solid lines denote data-packet flows.

in the XORP module handles inter-process communication between XORP and Click. All the processes and kernel threads constituting a SEATTLE switch are spawned and managed by a single master process.

The current OSPF implementation in XORP assumes each network node (router or switch) is identified only by an IP address, rather than a MAC address. Because of this, our prototype switch does not automatically select its identifier from the MAC addresses assigned to its interfaces. Instead, each switch is assigned a unique IP address on its loop-back interface, and advertises that address via a router LSA. For the same reason, when switches forward data packets across multiple hops (i.e., from ingress to egress or resolver), they use Ethernet-in-IP encapsulation [Housley and Hollenbeck 2002]. It is straightforward to extend our implementation to perform Ethernet-in-Ethernet encapsulation and ensure simplicity and self-configuration. We note that other link-state protocols such as IS-IS [Oran 1990] can work with non-IP interfaces, but XORP does not currently support these protocols.

When our prototype switch first starts up, a script detects the state of each network-interface card. It then executes a simple neighbor-discovery protocol to determine which interfaces are connected to other switches. Once all the switch-facing interfaces are correctly detected, the switch initiates an OSPF session and exchanges LSAs with its neighbors. The default link weight associated with the OSPF adjacency is determined based on the link's capacity (i.e., speed), and a different value may be used if desired. Meanwhile, the switch begins to receive packets through host-facing interfaces and learns end-hosts' information.

7.2 Building SEATTLE's Control Plane in Click

As previously mentioned, SEATTLE employs several control messages to handle host information, including host-information publication, notification for ingress caching, and updates due to host churn. We implemented this functionality as an element, named HostInfoExchanger, in the Click process because most SEATTLE control messages are directly triggered while handling data packets (e.g., MAC learning on the host-facing interface) except for the host re-publication due to topology changes.

In order to forward packets, the SEATTLE switch must know the location of destination hosts. To deal with this, we implemented the HostLocTable module. The HostLocTable is populated with three kinds of host information: (a) the physical interface (port) of every locally-attached host; (b) the location (adjacent switch's id) of every remote host for which this switch is a resolver; and (c) the location of every remote host cached for shortest-path forwarding. An insertion or deletion on this table takes place when a new host arrives, an existing host leaves, a host is published to or withdrawn from the switch, a host's information is retrieved from a resolver for ingress caching, or a cached host-information is evicted due to a cache-replacement decision. In addition, for each insertion or deletion of a locally-attached host, the SFE also communicates with the HostInfoExchanger and triggers a corresponding publication or withdrawal message. To maintain IP-to-MAC mappings for ARP, a switch also maintains a separate table in the control plane. While this table is similarly maintained to the HostLocTable, insertion and deletion operations on the table are triggered only by explicit publication and withdrawal respectively.

Finally, we also modified Click to monitor changes on the link-state map maintained by XORP. This is needed because switch failures or recoveries may require re-publication of host information. The Click process does this by monitoring the NextHopTable data structure, which stores every live switch in the network along with a corresponding next hop for each switch. The NextHopTable is updated by the XORP process whenever the network topology changes. Upon each change on the NextHopTable, the Click process also modifies its consistent hash ring, namely CHashRing, built with all switch identifiers found in the NextHopTable. This hash ring allows the Click data plane to perform consistent hashing and thus correctly map a host to its resolver. When the number of live switches in NextHopTable gets altered due to topology changes, the SFE first modifies the hash ring by adding or deleting the switch on the ring. If this change to the ring triggers re-publication of host information, the SFE makes the HostInfoExchanger issue corresponding control messages (i.e., publications or withdrawals), which are reliably exchanged via a simple acknowledgment-based protocol.

7.3 Building SEATTLE's Data Plane in Click

The way the SFE handles each packet is illustrated in Figure 7. Packet processing utilizes all four control-plane modules we mentioned above: the NextHopTable to determine a next hop, the CHashRing to map a host to a resolver, the HostLocTable to determine host location information, and the HostInfoExchanger to exchange host information with other switches.

When a packet is received on a host-facing interface, the switch first learns an incoming packet's source MAC address and looks up the corresponding entry in the HostLocTable. If there no corresponding entry is found, a new one is created and stored along with the interface on which the packet was received. Note that only the switch adjacent to the host performs this process because SEATTLE switches are not supposed to learn sources from frames received through switch-to-switch interfaces.

Then, the switch looks up the destination MAC address in the HostLocTable. If the lookup fails, the switch executes the consistent hash function $\mathcal{F}$ with the destination MAC address. It subsequently obtains a corresponding resolver switch's id, encapsulates the packet with that identifier, determines the next hop by looking up the NextHopTable with the resolver's id, and sends out the packet to the chosen next hop. On the other hand, if the lookup succeeds, the next action is determined based on three possible cases: the switch knows the destination host because (i) the destination is directly connected to the switch, (ii) the destination was published to the

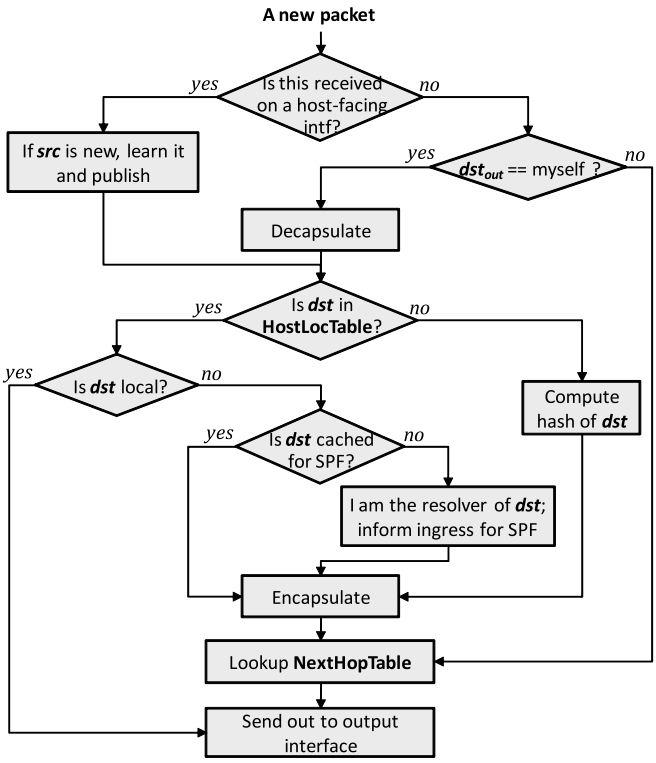


Fig. 7. Packet processing flowchart. **src** and **dst** respectively denote the source and destination MAC addresses in the original packet header, and $\mathbf{dst_{out}}$ denotes the destination address in the outer encapsulation header. SPF stands for shortest-path forwarding.

switch because the switch is the destination's resolver, or (iii) the destination's location was cached at the switch by earlier packets to the same destination. In case (i), the switch simply directs the packet to a corresponding output port. In cases (ii) or (iii), the switch encapsulates the packet with the identifier of the destination's adjacent switch (per the HostLocTable), determines the next hop (per the NextHopTable), and forwards the encapsulated packet to the next hop. Additionally for (ii), the switch must send a host-location notification message to the ingress for shortest-path forwarding.

Note that this host location resolution procedure takes place only at ingress, resolver, and egress switches. All the other intermediate switches simply forward packets based on the destination switch's id contained in the outer header. SEATTLE switches distinguish encapsulated packets from regular packets via the Protocol ID field in the IP header. In an implementation using Ethernet-in-Ethernet encapsulation, the EtherType field in the outer Ethernet header can provide the same functionality.

In addition, if the incoming packet is an ARP request, the ingress switch executes the hash function $\mathcal{F}$ to look up the corresponding resolver's id, and re-writes the destination to that id, and delivers the packet to the resolver for unicast ARP resolution.

8. EXPERIMENTAL RESULTS FROM EMULATION TESTS

In this section, we report performance results from a deployment of our prototype implementation on Emulab [White et al. 2002]. First, for cross-validation purposes, we compare the performance of our prototype switches with the simulation results

Table I. Cross Validation Results

	Stretch		**Table Size (# entries)**		**Control Overhead (# control messages)**	
	Emulation	Simulation	Emulation	Simulation	Emulation	Simulation
SEATTLE	1.03	1.00	3,896	3,979	17.2K	17.4K
SEATTLE (w/o ingress caching)	1.36	1.37	3,477	3,500	8.1K	8.3K
Ethernet	1.25	1.30	5,974	6,155	549.8K	561.6K

Table II. Per-Packet Processing Time in Micro-Sec

	learn source	*look up host table*	*encapsulation*	*look up nexthop table*	*decapsulation*	*Total*
SEATTLE-ingress	0.61	0.63	0.67	0.62	-	2.53
SEATTLE-egress	-	0.63	-	-	0.03	0.66
SEATTLE-intermediate	-	-	-	0.67	-	0.67
Ethernet	0.63	0.64	-	-	-	1.27

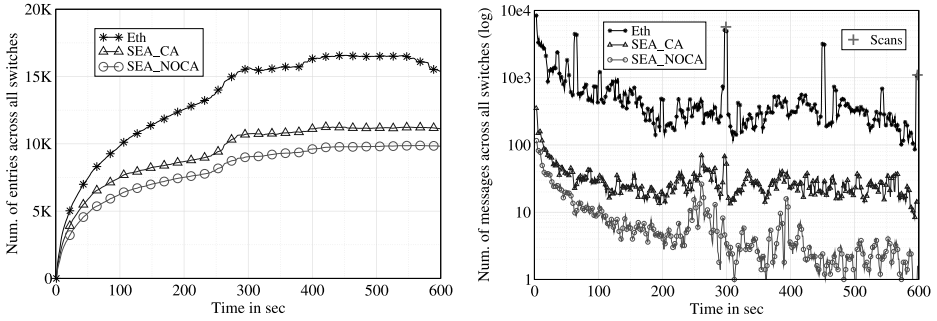
These numbers are measured using the Click-based SEATTLE implementation (explained in Section 7) and *EtherSwitch*, a publicly-available Ethernet bridge implementation included in the public Click distribution. We ran the Click process in the user space of a FreeBSD PC equipped with a 3.0 GHz processor and 2 GB of memory. The absolute numbers in this table are not particularly meaningful as they may change on different platforms. The difference between SEATTLE and Ethernet numbers, however, clearly distinguishes between the two technologies.

(shown in Section 6). Next, we present a set of microbenchmarks to evaluate per-packet processing overheads. Then, to evaluate dynamics of a SEATTLE network, we measure control overhead and switch state requirements over time. Finally, we investigate impact of SEATTLE on application-level performance, through a combination of web benchmarks.

For all experiments summarized in this section, we actually replayed the same traffic traces we used for the simulation tests in Section 6 using a set of hosts acting as traffic generators. Unless otherwise mentioned, the results shown in this section are from tests using the user-space SEATTLE implementation (i.e., a Click process running as a user process).

Cross-Validation. To ensure simulation results collected in the previous section would roughly characterize performance of a real SEATTLE deployment, we conducted experiments to cross-validate the simulator and implementation. We did this by configuring the simulator and implementation with identical traffic traces, topology, and protocol parameters. We then measured stretch, control overhead, and forwarding-table size for both Ethernet and SEATTLE in Table I. We found that average stretch, control overhead, and table size from implementation results were within 3% of the values given by the simulator. In general, we found these metrics exhibit similar trends under all the topologies and workloads we tried.

Packet Processing Overhead. Table II shows per-packet processing time for both SEATTLE and Ethernet. We measure this as the time from when a packet enters the switch's inbound queue, to the time it is ready to be moved to an outbound queue. We break this time down into the major components. From the table, we can see that SEATTLE switches exercise different sets of functional components based on their roles (ingress, egress, or intermediate) during packet delivery, whereas



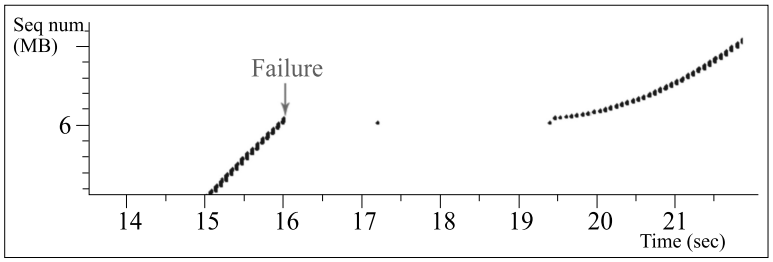
(a) Table size (b) Control overhead

Fig. 8. Time-series charts showing how SEATTLE and Ethernet react to network dynamics.

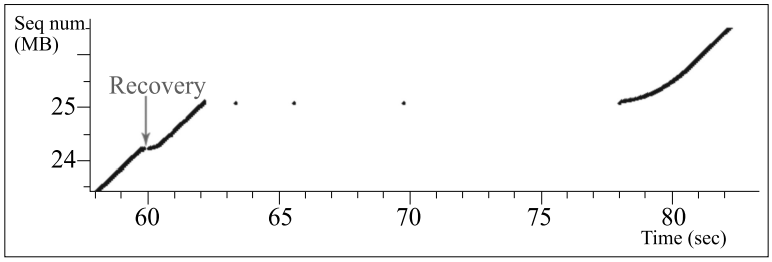
Ethernet bridges perform the same way all the time. In particular, an ingress switch in SEATTLE introduces more processing time than in Ethernet. This happens because the ingress switch has to encapsulate a packet and then look up the next-hop table with the outer header. However, SEATTLE introduces less packet processing overhead than Ethernet at noningress hops, as intermediate and egress switches do not need to learn source MAC addresses, and consistent hashing (which takes around 2.2 us on our test platform) is required only for ARP requests. Hence, SEATTLE introduces less overall processing time on paths longer than 3.03 switch-level hops. In comparison, we found the average number of switch-level hops between hosts in a real university campus network (*Campus*) to be over 4 for the vast majority of host pairs. Using our kernel-level implementation of SEATTLE, we were able to fully saturate a 1-Gbps link [Pall 2008].

Effect of Network Dynamics. To evaluate the dynamics of SEATTLE and Ethernet, we instrumented the switch's internal data structures to periodically measure performance information. Figures 8(a) and 8(b) show forwarding-table size and control overhead, respectively, measured over one-second intervals. We can see that SEATTLE has much lower control overhead when the systems first start up. However, SEATTLE's performance advantages do not come from cold-start effects, as it retains lower control overhead even after the system converges. As a side note, the forwarding-table size in Ethernet is not drastically larger than that of SEATTLE in this experiment because we are running on a small four-node topology. However, since the topology has ten links (including links to hosts), Ethernet's control overhead remains substantially higher. Additionally, we also investigate performance by injecting host scanning attacks [Allman et al. 2007] into the real traces we used for evaluation. Figure 8(b) includes the scanning incidences occurred at around 300 and 600 seconds, each of which involves a single host scanning 5000 random destinations that do not exist in the network. In Ethernet, every scanning packet sent to a destination generates a network-wide flood because the destination does not exist, resulting in sudden peaks on the control-overhead curve of Ethernet. In SEATTLE, each scanning packet generates one unicast lookup (i.e., the scanning data packet itself) to a resolver, which then discards the packet.

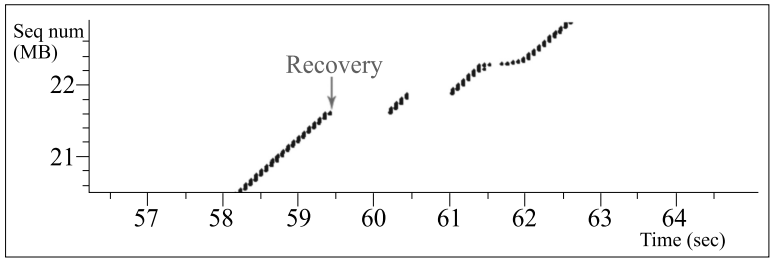
Fail-Over Performance. Figure 9 shows the effect of switch failure. To evaluate SEATTLE's ability to quickly republish host information, here we intentionally disable caching, induce failures of the resolver switch, and measure a TCP connection's behavior when all packets in the connection are forwarded through the resolver. We set the OSPF hello interval to 1 second, and dead interval to 3 seconds. After the



(a) Impact on a TCP flow by a switch failure



(b) Impact on a TCP flow by a switch recovery (w/o optimization)



(c) Impact on a TCP flow by a switch recovery (with optimization)

Fig. 9. Failover performance.

resolver fails, there is some convergence delay before packets are sent via the new resolver. We found that SEATTLE restores connectivity quickly, typically on the order of several hundred milliseconds after the dead interval. This allows TCP to recover within several seconds, as shown in Figure 9(a). We found performance during failures could be improved by having the access switch register hosts with the next switch along the ring in advance, avoiding an additional re-registration delay. When a switch is repaired, there is also a transient outage while routes move back over to the new resolver. Figure 9(b) shows TCP can recover in several seconds from switch repairs as well. Additionally, as shown in Figure 9(c), we were able to improve convergence delay during recoveries by letting switches continue to forward packets through the old resolver for a grace period. In contrast, optimizing Ethernet to attain low (a few seconds) convergence delay (by aggressively turning on or off switch ports to rapidly rebuild a spanning tree) exposes the network to a high chance of broadcast storms, making it nearly impossible to realize in a large network.

Impact on Application-Level Performance. We evaluated how SEATTLE's host-information resolution mechanism affects the application performance by running a

Table III. Web Performance Benchmarks: (Download time per file in sec.)

	SEATTLE			*Ethernet*		
	h1	h2	h3	h1	h2	h3
5KB	0.07	0.10	0.06	0.15	0.09	0.10
50KB	0.15	0.22	0.11	0.35	0.21	0.23
500KB	0.57	0.82	0.42	1.36	0.81	0.88

web benchmark. Here, we configured a topology consisting of four switches connected in a full mesh via links with 10 to 20 msec of delays. We placed four hosts *h0*, *h1*, *h2*, and *h3*, one at each of the four switches. To quantify the effect of the suboptimal forwarding for the first few packets, we emptied each switch's host-information cache without emptying the ARP cache of the four hosts and then ran the *Flexiclient* web benchmark tool [Pai et al. 1999] to generate a collection of test files and request patterns. Finally we measured the amount of time required for hosts *h1*, *h2*, and *h3* to download these files from *h0* using both SEATTLE and Ethernet. Table III shows the results with three different file sizes ranging from 5KB to 500KB over 50 trials. At first, one might think that SEATTLE – even with shortest-path forwarding – may increase download latency, as the first few packets may traverse a different path from later packets, possibly resulting in reordering. However, as can be seen from the table, this effect is negligible and has little effect on download time. This is because delivering first few packets in a connection has little impact on TCP's congestion-control and round-trip-time estimation logic. Note that the performance of Ethernet in the case of $h1$ and $h3$ is worse than that of SEATTLE because traffic between $h0$ and $h1$/$h3$ is forwarded through the root bridge $h2$.

9. RELATED WORK

The primary goal of our work is to design and implement a practical replacement for Ethernet that scales to large and dynamic networks. Although there are many approaches to improve Ethernet bridging, none of them are suitable for our purposes. RBridges [Perlman 2004; TRILL 2010] leverage a link-state protocol to disseminate bridge connectivity and employ a broadcast-based host-information dissemination scheme combined with source learning. Doing this eliminates the need to maintain a spanning tree and improves forwarding paths. CMU-Ethernet [Myers et al. 2004] also leverages link-state routing and replaces end-host broadcasting by propagating host information in link-state updates. Viking [Sharma et al. 2004] uses multiple spanning trees for faster fault recovery, which can be dynamically adjusted to conform to changing load. SmartBridges [Rodeheffer et al. 2000] allows shortest-path forwarding by obtaining the network topology, and monitoring which end host is attached to each switch. However, its control-plane overheads and storage requirements are similar to Ethernet, as every end host's information is disseminated to every switch. While SEATTLE was inspired by the problems addressed in these works, SEATTLE takes a radically different approach that eliminates network-wide dissemination of per-host information. This results in substantially improved control-plane scalability and data-plane efficiency. While there has been work, conducted in parallel with ours, that proposes hashing to support flat addressing [Ray et al. 2007], their design does not promptly handle host dynamics and forwards packets along a spanning tree, rather than through the shortest path. In contrast to this prior work, SEATTLE additionally enables some of the critical features especially useful for large networks, such as fault isolation, small lookup latency, and support for multiple small routing domains that can be managed independently.

Key differences of this article from our earlier work [Kim et al. 2008] include a detailed explanation of a prototype SEATTLE switch (Section 7), additional evaluation results using the prototype (Section 8), and a clearer description on the multilevel one-hop DHT design (Section 3.3) and a reachability isolation mechanism that improves the VLAN semantics (Section 5.2).

The design we propose is also substantially different from recent work on identity-based routing (ROFL [Caesar et al. 2006b], UIP [Ford 2004], and VRR [Caesar et al. 2006a]); our solution is suitable for building a practical and easy-to-manage network for several reasons. First, these previous approaches determine paths based on a hash of the destination's identifier (or the identifier itself), incurring a large stretch penalty which is unbounded in the worst case. In contrast, SEATTLE does *not* perform identity-based routing. Instead, SEATTLE uses resolution to map a MAC address to a host's location, and then uses the location to deliver packets along the shortest path to the host. This reduces latency and makes it easier to control and predict network behavior. Predictability and controllability are extremely important in real networks, because they make essential management tasks (e.g., capacity planning, troubleshooting, traffic engineering) possible. Second, the path between two hosts in a SEATTLE network does not change as other hosts join and leave the network. This substantially reduces packet reordering and improves constancy of path performance. Further, SEATTLE employs traffic-driven caching of host information, as opposed to the traffic-agnostic caching used in previous work (e.g., finger caches in ROFL). By only caching information that is actually needed to forward packets, SEATTLE significantly reduces the amount of state required to deliver packets. Finally, our design consists of several generic components, such as the multilevel one-hop DHT and service discovery mechanism, which could be adapted to the work in ROFL, UIP, and VRR.

SEATTLE also relates to other recent work on scalable network architectures for large data centers, including VL2 [Greenberg et al. 2009], PortLand [Mysore et al. 2009], and MOOSE [Scott and Crowcroft 2008]. While these designs propose separating hosts' flat names from their hierarchical locations to achieve scalability, agility, and ease of configuration, their technical approaches differ from SEATTLE. VL2 introduces a logically centralized, server-based (rather than network-based) directory system to maintain host information, along with host-information resolution and traffic-engineering mechanisms triggered and controlled by end hosts, rather than ingress switches. PortLand also employs a centralized directory system to maintain host information, but the host information itself is determined and assigned by distributed switches that discover their own topological positions by taking advantage of the multirooted tree topology prevalent in data centers. Centralizing network control may be possible for cloud-service data centers because a cloud-service provider often controls every host's lifetime and information via a logically centralized virtual-machine manager. Taking advantage of end-host capabilities to enhance a network is also a reasonable approach for cloud-service data centers where end-host environment is highly homogeneous due to virtualization. Unlike VL2 and PortLand, however, SEATTLE introduces an entirely decentralized highly-scalable solution, useful not only for cloud-service data centers or tree-based networks, but for any general large flat-addressing networks interconnecting autonomous and heterogeneous end hosts. We also compared SEATTLE's decentralized approach with other centralized approaches in Section 3.

10. CONCLUSION

Operators today face significant challenges in managing and configuring large networks. Many of these problems arise from the complexity of administering IP networks. Traditional Ethernet is not a viable alternative (except perhaps in small LANs)

due to poor scaling and inefficient path selection. We believe that SEATTLE takes an important first step towards solving these problems, by providing scalable self-configuring routing. Our design provides effective protocols that discover neighbors, resolve hosts' information, handle switch and host dynamics, deliver packets through shortest paths, and yet do not require any configuration for addressing and subnetting. Hence, in the simplest case, network administrators can construct an operational network without modifying any protocol settings. However, SEATTLE also provides add-ons for administrators who wish to customize network operation. Experiments with our initial prototype implementation show that SEATTLE provides efficient routing with low latency, quickly recovers after failures, and handles host mobility and network topology changes with low control overhead.

Moving forward, we are interested in improving the SEATTLE design for security. We are also interested in ramifications on switch architectures, and how to design switch hardware to efficiently support SEATTLE. Finally, to ensure deployability, this paper assumes networking stacks at end hosts are not modified. It would be interesting to consider what performance optimizations are possible if end host software can be changed.

REFERENCES

ADJIE-WINOTO, W., SCHWARTZ, E., BALAKRISHNAN, H., AND LILLEY, J. 1999. The design and implementation of an intentional naming system. In *Proceedings of the ACM Symposium on Operating Systems Principles*.

AIELLO, W., KALMANEK, C., MCDANIEL, P., SEN, S., SPATSCHECK, O., AND VAN DER MERWE, J. 2005. Analysis of communities of interest in data networks. In *Proceedings of the Symposium on Passive and Active Measurement*.

ALLMAN, M., PAXSON, V., AND TERRELL, J. 2007. A brief history of scanning. In *Proceedings of the Symposium on Internet Measurement Conference*.

ARREGOCES, M. AND PORTOLANI, M. 2003. *Data Center Fundamentals*. Cisco Press.

BARROSO, L. A. AND HOLZLE, U. 2009. *The Datacenter as a Computer*. Morgan & Claypool.

CAESAR, M., CASTRO, M., NIGHTINGALE, E., ROWSTRON, A., AND O'SHEA, G. 2006a. Virtual ring routing: Network routing inspired by DHTs. In *Proceedings of ACM SIGCOMM*.

CAESAR, M., CONDIE, T., KANNAN, J., LAKSHMINARAYANAN, K., AND STOICA, I. 2006b. ROFL: Routing on flat labels. In *Proceedings of ACM SIGCOMM*.

CASADO, M., FREEDMAN, M., PETTIT, J., LUO, J., GUDE, N., MCKEOWN, N., AND SHENKER, S. 2009. Rethinking enterprise network control. *IEEE/ACM Trans. Network*.

DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. 2001. Wide-area cooperative storage with CFS. In *Proceedings of the ACM Symposium on Operating Systems Principles*.

DEAN, J. AND GHEMAWAT, S. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Operating Systems Design and Implementation*.

DROMS, R. 1997. Dynamic host configuration protocol. RFC 2131.

FELDMEIER, D. C. 1988. Improving gateway performance with a routing-table cache. In *Proceedings of the IEEE INFOCOM*.

FORD, B. 2004. Unmanaged Internet Protocol: Taming the edge network management crisis. *ACM SIGCOMM Computer Comm. Rev*.

GODFREY, B., LAKSHMINARAYANAN, K., SURANA, S., KARP, R., AND STOICA, I. 2003. Load balancing in dynamic structured P2P systems. In *Proceedings of the IEEE INFOCOM*.

GRATUITOUS ARP. 2009. Gratuitous ARP - The Wireshark Wiki. http://wiki.wireshark.com/Gratuitous_ARP.

GREENBERG, A., HAMILTON, J., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D., PATEL, P., AND SENGUPTA, S. 2009. VL2: A scalable and flexible data center network. In *Proceedings of ACM SIGCOMM*.

GUPTA, A., LISKOV, B., AND RODRIGUES, R. 2004. Efficient routing for peer-to-peer overlays. In *Proceedings of the USENIX Networked Systems Design and Implementation*.

HALABI, S. 2003. *Metro Ethernet*. Cisco Press.

HANDLEY, M., KOHLER, E., GHOSH, A., HODSON, O., AND RADOSLAVOV, P. 2005. Designing extensible IP router software. In *Proceedings of the USENIX Networked Systems Design and Implementation*.

HEIMLICH, S. A. 1990. Traffic characterization of the NSFNET national backbone. In *Proceedings of ACM SIGMETRICS*.

HINDEN, R. 2004. Virtual router redundancy protocol (VRRP). RFC 3768.

HOUSLEY, R. AND HOLLENBECK, S. 2002. EtherIP: Tunneling ethernet frames in IP datagrams. RFC 3378.

HUCABY, D. AND MCQUERRY, S. 2002. *Cisco Field Manual: Catalyst Switch Configuration*. Cisco Press.

HUDSON, H. 2002. *Extending Access to the Digital Economy to Rural and Developing Regions. Understanding the Digital Economy*. The MIT Press, Cambridge, MA.

IEEE 802.1Q. 2005. IEEE Std 802.1Q–2005. IEEE standard for local and metropolitan area network, virtual bridged local area networks.

ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. 2007. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of EuroSys*.

JAIN, R. 1989/1990. Characteristics of destination address locality in computer networks: A comparison of caching schemes. *Comput. Netw. ISDN 18*, 243–254.

JAIN, R. AND ROUTHIER, S. 1986. Packet trains: Measurements and a new model for computer network traffic. *IEEE J. Select. Areas Comm*.

KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the ACM Symposium on Theory of Computing*.

KERRAVALA, Z. 2002. Configuration management delivers business resiliency. The Yankee Group.

KIM, C., CAESAR, M., AND REXFORD, J. 2008. Floodless in SEATTLE: A scalable ethernet architecture for large enterprises. In *Proceedings of ACM SIGCOMM*.

KIM, C., CAESAR, M., GERBER, A., AND REXFORD, J. 2009. Revisiting route caching: The world should be flat. In *Proceedings of the Symposium on Passive and Active Measurement*.

KING, R. 2004. Traffic management tools fight growing pains. http://www.thewhir.com/features/traffic-management.cfm.

KODIALAM, M., LAKSHMAN, T. V., AND SENGUPTA, S. 2004. Efficient and robust routing of highly variable traffic. In *Proceedings of the ACM Workshop on Hot Topics in Networks*.

KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. 2000. The Click modular router. *ACM Trans. Computer Syst*.

MYERS, A., NG, E., AND ZHANG, H. 2004. Rethinking the service model: Scaling Ethernet to a million nodes. In *Proceedings of the ACM Workshop on Hot Topics in Networks*.

MYSORE, R. N., PAMBORIS, A., FARRINGTON, N., HUANG, N., MIRI, P., RADHAKRISHNAN, S., AND SUBRAM, V. 2009. PortLand: A scalable fault-tolerant layer 2 data center network. In *Proceedings of ACM SIGCOMM*.

ORAN, D. 1990. OSI IS-IS Intra-domain routing protocol. RFC 1142.

PAI, V., DRUSCHEL, P., AND ZWAENEPOEL, W. 1999. Flash: An efficient and portable web server. In *Proceedings of the USENIX Annual Technical Conference*.

PALL, D. 2008. Faster packet forwarding in a scalable ethernet architecture. Tech. rep. TR-812-08, Computer Science Department, Princeton University.

PANG, R., ALLMAN, M., BENNETT, M., LEE, J., PAXSON, V., AND TIERNEY, B. 2005. A first look at modern enterprise traffic. In *Proceedings of the Internet Measurement Conference*.

PARTRIDGE, C. 1996. Locality and Route Caches. http://www.caida.org/workshops/isma/9602/positions/partridge.html.

PARTRIDGE, C., CARVEY, P. P., BURGESS, E., CASTINEYRA, I., CLARKE, T., GRAHAM, L., HATHAWAY, M., HERMAN, P., KING, A., KOHALMI, S., MA, T., MCALLEN, J., MENDEZ, T., MILLIKEN, W. C., PETTYJOHN, R., ROKOSZ, J., SEEGER, J., SOLLINS, M., STORCH, S., TOBER, B., AND TROXEL, G. D. 1998. A 50-Gb/s IP router. *IEEE/ACM Trans. Network 6*, 3.

PERLMAN, R. 1985. An algorithm for distributed computation of a spanning tree in an extended LAN. *ACM SIGCOMM Computer Comm. Rev*.

PERLMAN, R. 1999. *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols* 2nd Ed. Addison-Wesley.

PERLMAN, R. 2004. Rbridges: Transparent routing. In *Proceedings of IEEE INFOCOM*.

PLUMMER, D. C. 1982. An ethernet address resolution protocol – or – converting network protocol addresses to 48.bit ethernet address for transmission on ethernet hardward. RFC 826, Internet Standard 37.

RAY, S., GUERIN, R. A., AND SOFIA, R. 2007. A distributed hash table based address resolution scheme for large-scale Ethernet networks. In *Proceedings of the IEEE International Conference on Communications*.

RODEHEFFER, T., THEKKATH, C., AND ANDERSON, D. 2000. SmartBridge: A scalable bridge architecture. In *Proceedings of ACM SIGCOMM*.

SCOTT, M. AND CROWCROFT, J. 2008. MOOSE: Addressing the Scalability of Ethernet. In *Proceedings of EuroSys* (Poster session).

SHARMA, S., GOPALAN, K., NANDA, S., AND CHIUEH, T. 2004. Viking: A multi-spanning-tree Ethernet architecture for metropolitan area and cluster networks. In *Proceedings of the IEEE INFOCOM*.

SKITTER. 2010. CAIDA:tools:measurement:skitter. http://www.caida.org/tools/measurement/skitter.

SPRING, N., MAHAJAN, R., WETHERALL, D., AND ANDERSON, T. 2004. Measuring ISP topologies with Rocketfuel. *IEEE/ACM Trans. Network*.

TENGI, C., ROBERTS, J., CROUTHAMEL, J., MILLER, C., AND SANCHEZ, C. 2004. autoMAC: A tool for automating network moves, adds, and changes. In *Proceedings of the USENIX Large Installation System Administration Conference*.

TRILL 2010. IETF TRILL Working group. http://datatracker.ietf.org/wg/trill/.

VARGHESE, G. AND PERLMAN, R. 1990. Transparent interconnection of incompatible local area networks using bridges. *IEEE J. Select. Areas Comm.*

WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, G., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. 2002. An integrated experimental environment for distributed systems and networks. In *Proceedings of the Symposium on Operating Systems Design and Implementation*.

Received February 2010; revised July 2010; accepted September 2010