Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises

Changhoon Kim
Princeton University
Princeton, NJ
chkim@cs.princeton.edu

Matthew Caesar
University of Illinois
Urbana-Champaign, IL
caesar@cs.uiuc.edu

Jennifer Rexford
Princeton University
Princeton, NJ
jrex@cs.princeton.edu

ABSTRACT

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 paper 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 and stability, and simplifies network management. 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

Keywords

Enterprise network, Routing, Scalablity, Ethernet

1. INTRODUCTION

Ethernet stands as one of the most widely used networking technologies today. Due to its simplicity and ease of configuration, many enterprise and access provider networks utilize Ethernet as an elementary building block. Each host in an Ethernet is assigned a persistent MAC address, and Ethernet bridges automatically learn host addresses and locations. These "plug-and-play" semantics simplify many aspects of network configuration. Flat addressing simplifies the handling of topology changes and host mobility, without requiring administrators to reassign addresses.

However, Ethernet is facing revolutionary challenges. Today's layer-2 networks are being built on an unprecedented scale and with highly demanding requirements in terms of efficiency and availability. Large data centers are being built, comprising hundreds of thousands of computers within a single facility [1], and maintained by hundreds of network operators. To reduce energy costs, these data centers employ virtual machine migration and adapt to varying workloads, placing additional requirements on agility (e.g., host mobility, fast topology changes). Additionally, large metro Ethernet deployments contain over a million hosts and tens of thousands of bridges [2]. Ethernet is also being increasingly deployed in highly dynamic environments, such as backhaul for wireless campus networks, and transport for developing regions [3].

While an Ethernet-based solution becomes all the more important in these environments because it ensures service continuity and simplifies configuration, conventional Ethernet has some critical limitations. First, Ethernet bridging relies on network-wide *flooding* to locate end hosts. This results in large state requirements and control message overhead that grows with the size of the network. Second, Ethernet forces paths to comprise a *spanning tree*. Spanning trees perform well for small networks which 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*. This not only consumes excessive resources, but also 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

SIGCOMM'08, August 17–22, 2008, Seattle, Washington, USA.

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 Virtual LANs (VLANs) to build IP subnets independently of host location. While the overhead of address configuration and IP routing may be reduced by provisioning VLANs over a large number of, if not all, bridges, doing so reduces benefits of broadcast scoping, and worsens data-plane efficiency due to larger spanning trees. Efficiently assigning VLANs over bridges and links must also consider hosts' communication and mobility patterns, and hence is 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.

In this paper, 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 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 [4], which stores the *location* of each host. We use this network-layer DHT to build a flexible directory service which also performs *address resolution* (e.g., storing the MAC address associated with an IP address), and more flexible service discovery (e.g., storing the least loaded DNS server or printer within the domain). In addition, to provide stronger fault isolation and to support delegation of administrative control, we present a hierarchical, multi-level one-hop DHT.

Traffic-driven location resolution and caching: To forward packets along shortest paths and to avoid excessive load on the directory service, switches cache responses to queries. In enterprise networks, hosts typically communicate with a small number of other hosts [5], making caching highly effective. Furthermore, SEATTLE also provides a way to piggyback location information on ARP replies, which eliminates the need for location resolution when forwarding data packets. This allows data packets to directly traverse the shortest path, making the network's forwarding behavior predictable and stable.

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 reliable cache update protocol based on unicast. This ensures that all packets are delivered based on up-to-date state while keeping control overhead low. In contrast to conventional DHTs, this update process is directly triggered by network-layer changes, providing fast reaction times. For example, by observing link-state advertisements, switches determine when a host's location is no longer reachable, and evict those invalid entries. Through these approaches, SEATTLE seamlessly supports host mobility and other dynamics.

Despite these features, our design remains compatible with existing applications and protocols running at end hosts. For example, SEATTLE allows hosts to generate broadcast ARP and 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 trunk configuration.

1.1 Related work

Our quest is to design, implement, and evaluate a practical replacement for Ethernet that scales to *large and dynamic* networks. Although there are many approaches to enhance Ethernet bridging, none of these are suitable for our purposes. RBridges [6, 7] leverage a link-state protocol to disseminate information about both bridge connectivity and host state. This eliminates the need to maintain a spanning tree and improves forwarding paths. CMU-Ethernet [8] also leverages link-state and replaces end-host broadcasting by propagating host information in link-state updates. Viking [9] uses multiple spanning trees for faster fault recovery, which can be dynamically adjusted to conform to changing load. SmartBridges [10] 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. Though SEATTLE was inspired by the problems addressed in these works, it 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 on using hashing to support flat addressing conducted in parallel with our work [11, 12], these works do not promptly handle host dynamics, require some packets to be detoured away from the shortest path or be forwarded along a spanning tree, and do not support hierarchical configurations to ensure fault/path isolation and the delegation of administrative control necessary for large networks.

The design we propose is also substantially different from recent work on identity-based routing (ROFL [13], UIP [14], and VRR [15]). 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 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. Finally, SEATTLE employs traffic-driven caching of host information, as opposed to the traffic-agnostic caching (e.g., finger caches in ROFL) used in previous works. By only caching information that is needed to forward packets, SEATTLE significantly reduces the amount of state required to deliver packets. However, our design also consists of several generic components, such as the multi-level one-hop DHT and service discovery mechanism, that could be adapted to the work in [13, 14, 15].

Roadmap: We summarize how conventional enterprise networks are built and motivate our work in Section 2. Then we describe our main contributions in Sections 3 and 4 where we introduce a very simple yet highly scalable mechanism that enables shortest-path forwarding while maintaining the same semantics as Ethernet. In Section 5, we enhance existing Ethernet mechanisms to make our design backwards-compatible with conventional Ethernet. We then evaluate our protocol using simulations in Section 6 and an implementation in Section 7. Our results show that SEATTLE scales to networks containing two orders of magnitude more hosts than a traditional Ethernet network. As compared with ROFL, SEAT-

TLE 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.

2. TODAY'S ENTERPRISE AND ACCESS NETWORKS

To provide background for the remainder of the paper, 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[1]. Ethernet *bridges* are used to interconnect multiple segments into a multi-hop network, namely a LAN, forming a single *broadcast domain*. 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 multiple paths in the topology can lead to *broadcast storms*, where frames are repeatedly replicated and forwarded along a loop. To avoid this, bridges in a broadcast domain coordinate to compute a *spanning tree* [16]. Administrators first select and configure a single *root bridge*; then, the bridges collectively compute a spanning tree based on distances to the root. Links not present in the tree are not used to carry traffic, causing longer paths and inefficient use of resources. Unfortunately, Ethernet-bridged networks cannot grow to a large scale due to 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 proportionally 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 [17].

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. 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.

Dependence on broadcasting for basic operations: DHCP and ARP 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* handled by the network interface card, and instead must interrupt the CPU [18]. 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 for malicious hosts as they can easily launch ARP or DHCP floods [8].

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. 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 namespace, and must consider future use of addresses to minimize later reassignment. 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 massive 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 [19]. 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. DHCP automates end-host configuration, but introduces substantial 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 [20], 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 network design changes to prevent reachability problems and to avoid vulnerabilities. Ideally, administrators should only need to update policy configurations when the *policy* itself, not the *network,* changes.

[1] In modern switched Ethernet networks, a segment is just a point-to-point link connecting an end host and a bridge, or a pair of bridges.

Limited mobility support: Supporting seamless host mobility is becoming increasingly important. In data centers, migratable virtual machines are being widely deployed to improve power efficiency by adapting to workload, and to minimize service disruption during maintenance operations. 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 policy update overhead, 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 the same networking requirements 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, as hosts may retain their IP addresses while moving between bridges in the same VLAN. This also reduces policy reconfiguration overhead. Unfortunately, VLANs introduces several problems:

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 be in a given VLAN must consider traffic and 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 visible to 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), an expensive and power-intensive technology.

Insufficient data-plane efficiency: Larger enterprises and data centers often have richer topologies, for greater reliability and performance. Unfortunately, a single spanning tree is used in each VLAN to forward packets, which prevents certain links from being used. Although configuring a disjoint spanning tree for each VLAN [9, 21] may improve load balance and increase aggregate throughput, effective use of per-VLAN trees requires periodically moving the roots and rebalancing 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 conventional Ethernet is to route packets to a destination specified by a MAC address. To do this, Ethernet bridges collectively provide end hosts with a service that maps MAC addresses to physical locations. Each bridge implements this service by maintaining next-hop pointers associated with MAC addresses in its forwarding table, and relies on domain-wide flooding to keep these pointers up to date. Additionally, Ethernet also allows hosts to look up the MAC address associated with a given IP address by broadcasting *Address Resolution Protocol* (ARP) messages.

In order to provide the same interfaces to end hosts as conventional Ethernet, SEATTLE also needs a mechanism that maintains mappings between MAC/IP addresses and locations. To scale to large networks, SEATTLE operates a distributed directory service built using a *one-hop, network-level DHT*. We use a *one-hop* DHT to reduce lookup complexity and simplify certain aspects of network administration such as traffic engineering and troubleshooting. We use a *network-level* approach that stores mappings at switches, so as to ensure fast and efficient reaction to network failures and recoveries, and avoid the control overhead of a separate directory infrastructure. Moreover, our network-level approach allows storage capability to increase naturally with network size, and exploits *caching* to forward data packets directly to the destination without needing to traverse any intermediate DHT hops [22, 23].

3.1 Scalable key-value management with a one-hop DHT

Our distributed directory has two main parts. First, running a link-state protocol ensures each switch can observe all other switches in the network, and allows any switch to route any other switch along shortest paths. Second, SEATTLE uses a *hash function* to map host information to a switch. This host information 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 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 [8, 6, 10, 7], 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 the link-state information to compute shortest paths for unicasting, and multicast trees for 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 attached to other switches. Distinguishing between these different kinds of links is done by sending control messages that Ethernet hosts do not respond to. This process is similar to how Ethernet distinguishes switches from hosts when building its spanning tree. To identify themselves in the link-state protocol, SEATTLE switches determine their own unique *switch IDs* without administrator involvement. 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 learn this information in an on-demand fashion, via a simple hashing mechanism. This information is stored in the form of $(\text{key}= k, \text{value}= v)$ pairs. A *publisher* switch s_a wishing to publish a (k, v) pair via the directory service uses a hash function $\mathcal{F}$ to map 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

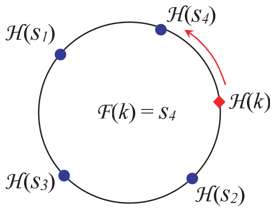


Figure 1: Keys are consistently hashed onto resolver switches (s_i).

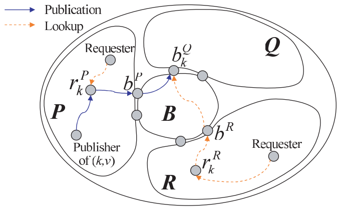


Figure 2: Hierarchical SEATTLE hashes keys onto *regions*.

knows all the other switches' identifiers via link-state advertisements from the routing protocol, and hence $\mathcal{F}$ works identically across 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 re-hashed to different resolver switches. To minimize this re-hashing overhead, SEATTLE utilizes *Consistent Hashing* [24] 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 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 the switch with the closest identifier not exceeding that of the key on the hash space of $\mathcal{H}$. As an optimization, a key may be additionally mapped to the next m closest switches along the hash ring, to improve resilience to multiple failures. However, in our evaluation, we will assume this optimization is disabled by default.

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 service the same load as more powerful switches. To deal with this, we propose a new 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 [23], it is *not* necessary for our virtual switches to be advertised to other physical switches. To reduce size of link-state advertisements, instead of advertising every virtual switch in the link-state protocol, switches only advertise the number of virtual switches they are currently running. Each switch then locally computes virtual switch IDs using the following technique. 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 $1 \leq i \leq L$. Note that it is possible to automate determining a desirable number of virtual switches per physical switch [25].

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"* to a switch, at which it may store its location or address information. Other switches can then reach the printer using the hash of the 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. Services can be named using techniques shown in previous work [26].

3.2 Responding to topology changes

The switch-level topology may 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.

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 switches in the network, the resolver a particular key maps to 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.

However, if a switch fails or recovers, the set of switches in the link-state map changes. Hence there may be some keys k whose old resolver r_k^{old} differs from a new resolver r_k^{new}. To deal with this, the value (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 monitor the liveness of k's resolver through link-state advertisements. 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. Additionally, when a value v denotes a location, such as a switch id s, and s goes down, each switch scans the list of locally-stored (k, v) pairs, and remove 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 Supporting hierarchy with a multi-level, one-hop DHT

The SEATTLE design presented so far scales to large, dynamic networks [27]. However, since this design runs a single, network-wide link-state routing protocol, it may be inappropriate for networks with highly dynamic infrastructure, such as networks in developing regions [3]. A single network-wide protocol may also be inappropriate if network operators wish to provide stronger fault isolation across geographic regions, or to divide up administrative control across smaller routing domains. Moreover, when a SEATTLE network is deployed over a wide area, the resolver could lie far both from the source and destination. Forwarding lookups over long distances increases latency and makes the lookup more prone to failure. To deal with this, SEATTLE may be configured hierarchically, by leveraging a *multi-level, one-hop DHT*. This mechanism is illustrated in Figure 2.

A hierarchical network is divided into several *regions*, and a *backbone* providing connectivity across regions. Each region is connected to the backbone via its own *border switch*, and the back-

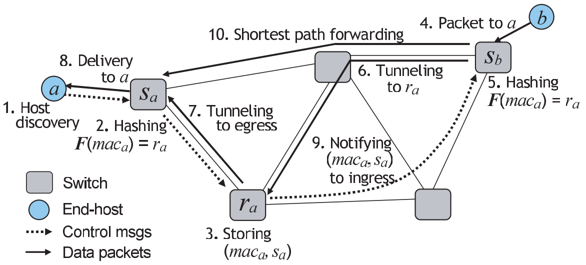


Figure 3: Packet forwarding and lookup in SEATTLE.

bone is composed of the border switches of all regions. Information about regions is summarized and propagated in a manner similar to *areas* in OSPF. In particular, each switch in a region knows the identifier of the region's border switch, because the border switch advertises its role through the link-state protocol. In such an environment, SEATTLE ensures that only inter-region lookups are forwarded via the backbone while all regional lookups are handled within their own regions, and link-state advertisements are only propagated locally within regions. SEATTLE ensures this by defining a separate *regional* and *backbone* hash ring. When a (k, v) is inserted into a region P and is published to a regional resolver r_k^P (i.e., a resolver for k in region P), r_k^P additionally forwards (k, v) to one of the region P's border switches b^P. Then b^P hashes k again onto the backbone ring and publishes (k, v) to another backbone switch b_k^Q, which is a backbone resolver for k and a border switch of region Q at the same time. Switch b_k^Q stores k's information. If a switch in region R wishes to lookup (k, v), it forwards the lookup first to its local resolver r_k^R, which in turn forwards it to b^R, and b^R forwards it to b_k^Q. As an optimization to reduce load on border switches, b_k^Q may hash k and store (k, v) at a switch within its own region Q, rather than storing (k, v) locally. Since switch failures are not propagated across regions, each publisher switch periodically sends probes to backbone resolvers that lie outside of its region. To improve availability, (k, v) may be stored at multiple backbone resolvers (as described in Section 3.1.2), and multiple simultaneous lookups may be sent in parallel.

4. SCALING ETHERNET WITH A ONE-HOP DHT

The previous section described the design of a distributed network-level directory service based on a one-hop DHT. In this section, we describe how the directory service is used to provide efficient packet delivery and scalable address resolution. 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 eliminate the need to 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 s_b then and 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 on towards s_a. In order to limit the number of data packets traversing the resolver, r_a also notifies s_b that a's current location is s_a. Switch s_b then caches this information. While forwarding the first few packets of a flow via a resolver switch increases path lengths, in the next section we describe an optimization that allows data packets to traverse only shortest paths, by piggy-backing location information on ARP replies.

Note SEATTLE manages per-host information via reactive resolution, as opposed to the proactive dissemination scheme used in previous approaches [8, 6, 10]. The scaling benefits of this reactive resolution increase in enterprise/data-center/access provider networks because most hosts communicate with a small number of popular hosts, such as mail/file/Web servers, printers, VoIP gateways, and Internet gateways [5]. To prevent forwarding tables from growing unnecessarily large, the access switches can apply various cache-management policies. For correctness, however, the cache-management scheme must not evict the host information of the 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. 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 with 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 broadcast-based ARP operations. 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.

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 the 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 [2]. Note that, even if the cache and the host become out of sync (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

Hosts can undergo three different 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 handoff), if its link has been plugged into a different access switch, or if it is a virtual machine and has migrated to a new hosting system that allows the VM to retain its MAC address. Second, a host may change its MAC address, for example if its NIC card is replaced, if it is a VM and has migrated to a new hosting system that requires the VM to use the host's MAC address, or if multiple physical machines collectively acting as a single server or router (to ensure high availability) experience a fail-over event [28]. Third, a host may change its IP address, for example if a DHCP lease expires, or if the host is manually reconfigured. In practice, multiple of these changes may occur simultaneously. When these changes occur, we need to keep the directory service up-to-date, to ensure correct packet delivery.

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 updates the value v associated with a given key k. First, in the case of a location change, 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 must be kept up-to-date as hosts move. SEATTLE 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 misdelivered packets of h's new location s_h^{new}. To minimize service disruption, s_h^{old} also forwards those misdelivered packets s_h^{new}.

Updating remote hosts' caches: 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, the new mapping (ip_h, mac_h^{new}) must be immediately known to other hosts who happened to store (ip_h, mac_h^{old}) in their local ARP caches. In conventional Ethernet, this is achieved by broadcasting a *gratuitous ARP request* originated by h [29]. A gratuitous ARP is an ARP request containing the MAC and IP address of the host sending it. 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 source or 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 to updating remote ARP caches does not require s_h to look up each packet's IP and MAC address pair from the revocation list because s_h can skip the lookup in the common case (i.e., when its revocation list is empty). Entries from the revocation list are removed after a timeout set equal to the ARP cache timeout of end hosts.

5. PROVIDING ETHERNET-LIKE SEMANTICS

To be fully backwards-compatible with conventional Ethernet, SEATTLE must act like a conventional Ethernet from the perspective of end hosts. First, the way that hosts interact with the network to bootstrap themselves (e.g., acquire addresses, allow switches to discover their presence) must be the same as Ethernet. Second, switches have to support traffic that uses broadcast/multicast Ethernet addresses as destinations. In this section, we describe how to perform these actions without incurring the scalability challenges of traditional Ethernet. For example, we propose to eliminate broadcasting from the two most popular sources of broadcast traffic: ARP and DHCP. Since we described how SEATTLE switches handle ARP without broadcasting in Section 4.2, we discuss only DHCP in this section.

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, except that MAC learning is enabled only on the ports connected to end hosts. 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 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 [29]. If a host does not generate a gratuitous ARP, the switch can still learn of the host's IP address via snooping on DHCP messages, or sending out an ARP request only on

[2] The default setting of the ARP cache timeout in most common operating systems ranges 10 to 20 minutes.

the port connected to the host. 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 [30]. 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 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 Scalable and flexible VLANs

SEATTLE completely eliminates flooding of unicast packets. However, to offer the same semantics as Ethernet bridging, SEATTLE needs to support transmission of packets sent to a *broadcast address*. Supporting broadcasting is important because some applications (e.g., IP multicast, peer-to-peer file sharing programs, etc.) rely on subnet-wide broadcasting. However, in large networks to which our design is targeted, performing broadcasts in the same style as Ethernet may significantly overload switches and reduce data plane efficiency. Instead, SEATTLE provides a mechanism which is similar to, but more flexible than, VLANs.

In particular, SEATTLE introduces a notion of *group*. Similar to a VLAN, a group is defined as a set of hosts who share the same broadcast domain regardless of their location. Unlike Ethernet bridging, however, a broadcast domain in SEATTLE does not limit unicast layer-2 reachability between hosts because a SEATTLE switch can resolve any host's address or location without relying on broadcasting. Thus, groups provide several additional benefits over VLANs. First, groups do not need to be manually assigned to switches. A group is automatically extended to cover a switch as soon as a member of that group arrives at the switch[3]. 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 allowed (or restricted) depending on the access-control policy — a rule set defining which groups can communicate with which — between the groups.

The flexibility of groups ensures several benefits that are hard to achieve with conventional Ethernet bridging and VLANs. When a group is aligned with a subnet, and unicast reachability between two different groups is not permitted by default, groups provide exactly the same functionality as VLANs. However, groups can include a large number of end hosts and can be extended to anywhere in the network without harming control-plane scalability and data-plane efficiency. Moreover, when groups are defined as subsets of an IP subnet, and inter-group reachability is prohibited, each group is equivalent to a private VLAN (PVLAN), which are popularly used in hotel/motel networks [32]. Unlike PVLANs, however, groups can be extended over multiple bridges. Finally, when unicast reachability between two groups is allowed, traffic between the groups takes the shortest path, without traversing default gateways.

Multicast-based group-wide broadcasting: Some applications may rely on subnet-wide broadcasting. To handle this, all broadcast packets within a group are delivered through a multicast tree sourced at a dedicated switch, namely a *broadcast root*, of the group. The mapping between a group and its broadcast root is determined by using $\mathcal{F}$ to hash the group's identifier to a switch. Construction of the multicast tree is done in a manner similar to IP multicast, inheriting its safety (i.e., loop freedom) and efficiency (i.e., to receive broadcast only when necessary). 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 tree toward g's broadcast root. When a host departs, its access switch prunes a branch if necessary. When an end host in g sends a broadcast packet, its access switch marks the packet with g and forwards it along g's multicast tree.

Separating unicast reachability from broadcast domains: In addition to handling broadcast traffic, groups in SEATTLE also provide a namespace upon which reachability policies for unicast traffic are defined. When a host arrives at an access switch, the host's group membership is determined by its access switch and published to the host's resolvers along with its location information. Access control policies are then applied by a resolver when a host attempts to look up a destination host's information.

6. SIMULATIONS

In this section, we start by describing our simulation environment. Next, we describe SEATTLE's performance under workloads collected from several real operational networks. We then investigate SEATTLE's performance in dynamic environments by generating host mobility and topology changes.

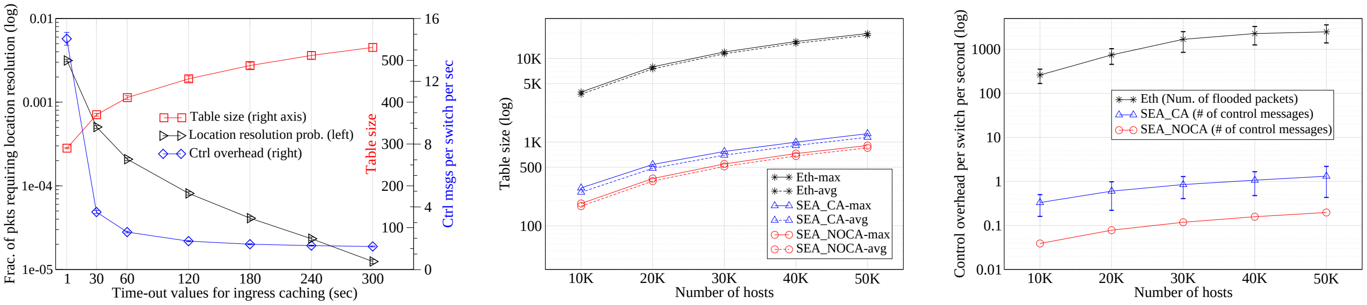
6.1 Methodology

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, network administrators (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 by Pang et. al. [33]. 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 [17].

We measured SEATTLE's performance on four representative

[3] The way administrators associate a host with corresponding group is beyond the scope of this paper. For Ethernet, management systems that can automate this task (e.g., mapping an end host or flow to a VLAN) are already available [31], and SEATTLE can employ the same model.



(a) (b) (c)

Figure 4: (a) Effect of cache timeout in *AP-large* with $50K$ hosts (b) Table size increase in *DC* (b) Control overhead in *AP-large*. These figures contain error bars showing confidence intervals for each data point. A sufficient number of simulation runs reduced these intervals.

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 [34]. Because SEATTLE switches are intended to replace both IP routers and Ethernet bridges, the routers in these topologies are considered as 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 [1].

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 [35] 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, 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.

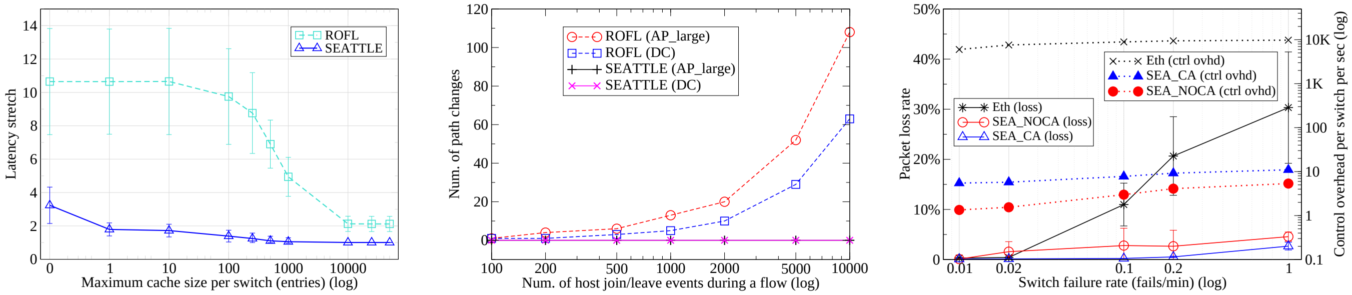
Given a fixed topology, the performance of SEATTLE and Ethernet bridging can vary depending on traffic patterns. To quantify this variation we repeated each simulation run 25 times, and plot the average of these runs with 99% confidence intervals. For each run we vary a random seed, causing the number 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.

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 4a 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 4b 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 drastically 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*.

Control overhead: Figure 4c shows the amount of control overhead generated by SEATTLE and Ethernet. We computed this value by dividing the total number of control messages 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. In general, we found that the difference in control overhead increased roughly with the number of links in the network.



(a) (b) (c)

Figure 5: (a) Stretch across different cache sizes in *AP-large* with $10K$ hosts (b) Path stability (c) Effect of switch failures in *DC*.

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 [15] 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 5a shows the average relative latency penalty, or *stretch*, of SEATTLE and ROFL [13] 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., every host communicates with all other hosts, switches maintain very small caches), SEATTLE still ensures reasonably small stretch. Finally, we compare *path stability* with ROFL in Figure 5b. 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) 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 5c shows performance during switch failures. Here, we cause switches to fail randomly, with failure inter-arrival times drawn from a Pareto distribution with $\alpha = 2.0$ and varying mean values. Switch recovery times are drawn from the same distribution, with a mean of 30 seconds. 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, using caching improved availability further.

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 6). 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 broadcast overhead by updating host state via unicasts.

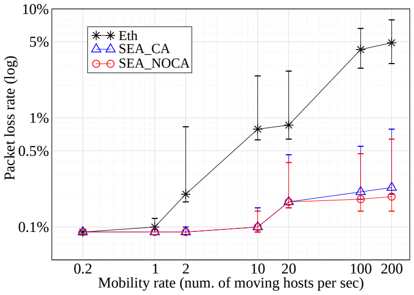


Figure 6: Effect of host mobility in *Campus*.

7. IMPLEMENTATION

To verify SEATTLE's performance and practicality through a real deployment, we built a prototype SEATTLE switch using two open-source routing software platforms: user-level *Click* [36] and *XORP* [37]. We also implemented a second version of our prototype using kernel-level Click [38]. Section 7.1 describes the structure of our design, and Section 7.2 presents evaluation results.

7.1 Prototype design

Figure 7 shows the overall structure of our implementation. SEATTLE's control plane is divided into two functional modules: *i*) maintaining the switch-level topology, and *ii*) managing end-host information. We used XORP to realize the first functional module, and used Click to implement the second. We also extended Click to implement SEATTLE's data-plane functions, including consistent hashing and packet encapsulation. Our control and data plane modifications to Click are implemented as the *SeattleSwitch* element shown in Figure 7.

SEATTLE control plane: First, we run a XORP OSPF process at each switch to maintain a complete switch-level network map. The XORP RIBD (Routing Information Base Daemon) constructs its routing table using this map. RIBD then installs the routing table into the forwarding plane process, which we implement with Click. Click uses this table, namely *NextHopTable*, to determine a next hop. The FEA (Forwarding Engine Abstraction) in XORP handles inter-process communication between XORP and Click. To maintain host information, a SeattleSwitch utilizes a *HostLocTable*, which is populated with three kinds of host information: (a) the outbound port for every local host; (b) the location for every remote host for which this switch is a resolver; and (c) the location for ev-

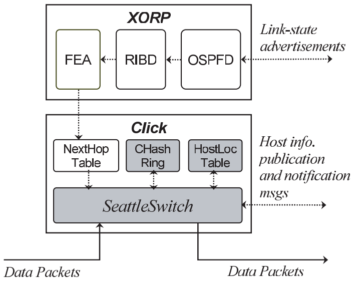


Figure 7: Implementation architecture.

ery remote host cached via previous lookups. For each insertion or deletion of a locally-attached host, the switch generates a corresponding registration or deregistration message. Additionally, by monitoring the changes of the NextHopTable, the switch can detect whether the topology has changed, and host re-registration is required accordingly. To maintain IP-to-MAC mappings to support ARP, a switch also maintains a separate table in the control plane. This table contains only the information of local hosts and remote hosts that are specifically hashed to the switch. When our prototype switch is first started up, a simple neighbor-discovery protocol is run to determine which interfaces are connected to other switches, and over each of these interfaces it initiates an OSPF session. The link weight associated with the OSPF adjacency is by default set to be the link latency. If desired, another metric may be used.

SEATTLE data plane: To forward packets, an ingress switch first learns an incoming packet's source MAC address, and if necessary, adds the corresponding entry in HostLocTable. Then the switch looks up the destination MAC address in the HostLocTable and checks to see if *i*) the host is locally attached, *ii*) the host is remote, and its location is cached, or *iii*) the host is explicitly registered with the switch. In the case of *iii*) the switch needs to send a host location notification to the ingress. In all cases, the switch then forwards the packet either to the locally attached destination, or encapsulates the packet and forwards it to the next hop toward the destination. Intermediate switches can then simply forward the encapsulated packet by looking up the destination in their NextHopTables. 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 resolution.

7.2 Experimental results

Next, we evaluate a deployment of our prototype implementation on Emulab. To ensure correctness, we cross-validated the simulator and implementation with various traces and topologies, and found that average stretch, control overhead, and table size from implementation results were within 3% of the values given by the simulator. We first 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, and evaluate switch fail-over performance.

Packet processing overhead: Table 1 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 an ingress switch in SEATTLE requires 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 requires less packet processing overhead than Ethernet at non-ingress hops, as intermediate and egress switches do not need to learn source MAC addresses, and consistent hashing (which takes around 2.2 us) is required only for ARP requests. Hence, SEATTLE requires 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.

Table 1: Per-packet processing time in micro-sec.

	learn src	*look-up host tbl*	*encap*	*look-up nexthop tbl*	*Total*
SEA-ingress	0.61	0.63	0.67	0.62	2.53
SEA-egress	-	0.63	-	-	0.63
SEA-others	-	-	-	0.67	0.67
ETH	0.63	0.64	-	-	1.27

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 8a and 8b 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 are first started 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 [17] into the real traces we used for evaluation. Figure 8b 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 is not existing, resulting in sudden peaks on it's control overhead curve. 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 8c 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 throughput of TCP when all packets are forwarded through the resolver. We set the OSPF hello interval to 1 second, and dead interval to 3 seconds. After the 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 8c-i. 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, as shown in Figure 8c-ii. In particular, 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 sec) convergence delay exposes the network to a high chance of broadcast storms, making it nearly impossible to realize in a large network.

8. 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

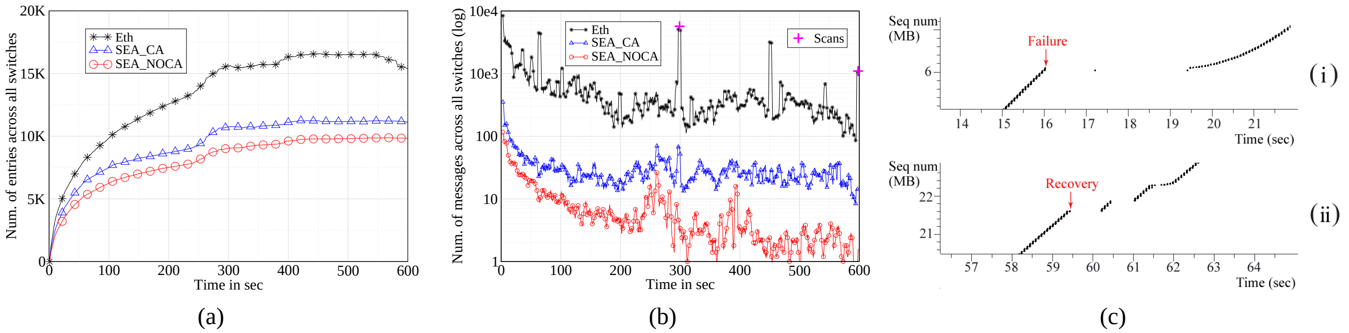


Figure 8: *Effect of network dynamics:* (a) table size (b) control overhead (c) failover performance.

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 to discover neighbors and operates efficiently with its default parameter settings. Hence, in the simplest case, network administrators do not need to modify 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 churn with low control overhead.

Moving forward, we are interested in investigating the deployability of SEATTLE. 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 Ethernet 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.

9. REFERENCES

[1] M. Arregoces and M. Portolani, *Data Center Fundamentals*. Cisco Press, 2003.

[2] S. Halabi, *Metro Ethernet*. Cisco Press, 2003.

[3] H. Hudson, “Extending access to the digital economy to rural and developing regions.” Understanding the Digital Economy, The MIT Press, Cambridge, MA, 2002.

[4] A. Gupta, B. Liskov, and R. Rodrigues, “Efficient routing for peer-to-peer overlays,” in *NSDI*, March 2004.

[5] W. Aiello, C. Kalmanek, P. McDaniel, S. Sen, O. Spatscheck, and J. van der Merwe, “Analysis of communities of interest in data networks,” in *Passive and Active Measurement*, March 2005.

[6] R. Perlman, “Rbridges: Transparent routing,” in *INFOCOM*, March 2004.

[7] “IETF TRILL working group.” `http://www.ietf.org/html.charters/trill-charter.html`.

[8] A. Myers, E. Ng, and H. Zhang, “Rethinking the service model: scaling Ethernet to a million nodes,” in *HotNets*, November 2004.

[9] S. Sharma, K. Gopalan, S. Nanda, and T. Chiueh, “Viking: A multi-spanning-tree Ethernet architecture for metropolitan area and cluster networks,” in *INFOCOM*, March 2004.

[10] T. Rodeheffer, C. Thekkath, and D. Anderson, “SmartBridge: A scalable bridge architecture,” in *SIGCOMM*, August 2000.

[11] C. Kim and J. Rexford, “Revisiting Ethernet: Plug-and-play made scalable and efficient,” in *IEEE LANMAN*, June 2007. invited paper (short workshop paper).

[12] S. Ray, R. A. Guerin, and R. Sofia, “A distributed hash table based address resolution scheme for large-scale Ethernet networks,” in *International Conference on Communications*, June 2007.

[13] M. Caesar, T. Condie, J. Kannan, *et al.*, “ROFL: Routing on Flat Labels,” in *SIGCOMM*, September 2006.

[14] B. Ford, “Unmanaged Internet Protocol: Taming the edge network management crisis,” in *HotNets*, November 2003.

[15] M. Caesar, M. Castro, E. Nightingale, *et al.*, “Virtual Ring Routing: Network routing inspired by DHTs,” in *SIGCOMM*, September 2006.

[16] R. Perlman, *Interconnections: Bridges, routers, switches, and internetworking protocols*. Addison-Wesley, second ed., 1999.

[17] M. Allman, V. Paxson, and J. Terrell, “A brief history of scanning,” in *Internet Measurement Conference*, October 2007.

[18] Dartmouth Institute for Security Technology Studies, “Problems with broadcasts,” `http://www.ists.dartmouth.edu/classroom/crs/arp_broadcast.php`.

[19] Z. Kerravala, “Configuration management delivers business resiliency,” November 2002. The Yankee Group.

[20] R. King, “Traffic management tools fight growing pains,” June 2004. `http://www.thewhir.com/features/traffic-management.cfm`.

[21] “IEEE Std 802.1Q - 2005, IEEE Standard for Local and Metropolitan Area Network, Virtual Bridged Local Area Networks,” 2005.

[22] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana, “Internet indirection infrastructure,” in *SIGCOMM*, August 2002.

[23] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, “Wide-area cooperative storage with CFS,” in *SOSP*, October 2001.

[24] D. Karger, E. Lehman, T. Leighton, *et al.*, “Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web,” in *ACM STOC*, 1997.

[25] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, “Load balancing in dynamic structured P2P systems,” in *INFOCOM*, March 2003.

[26] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, “The design and implementation of an intentional naming system,” in *SOSP*, December 1999.

[27] J. Moy, *OSPF: Anatomy of an Internet Routing Protocol*. Addison-Wesley, 1998.

[28] R. Hinden, “Virtual Router Redundancy Protocol (VRRP).” RFC 3768, April 2004.

[29] “Gratuitous ARP.” `http://wiki.ethereal.com/Gratuitous_ARP`.

[30] R. Droms, “Dynamic Host Configuration Protocol.” Request for Comments 2131, March 1997.

[31] C. Tengi, J. Roberts, J. Crouthamel, C. Miller, and C. Sanchez, “autoMAC: A tool for automating network moves, adds, and changes,” in *LISA Conference*, 2004.

[32] D. Hucaby and S. McQuerry, *Cisco Field Manual: Catalyst Switch Configuration*. Cisco Press, 2002.

[33] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney, “A first look at modern enterprise traffic,” in *Internet Measurement Conference*, October 2005.

[34] N. Spring, R. Mahajan, and D. Wetherall, “Measuring ISP topologies with Rocketfuel,” in *SIGCOMM*, August 2002.

[35] “Skitter.” `http://www.caida.org/tools/measurement/skitter`.

[36] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek, “The Click modular router,” in *ACM Trans. Comp. Sys.*, August 2000.

[37] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov, “Designing extensible IP router software,” in *NSDI*, May 2005.

[38] D. Pal, “Faster Packet Forwarding in a Scalable Ethernet Architecture.” TR-812-08, Princeton University, January 2008.