

Building Scalable Self-configuring Networks with SEIZE

Changhoon Kim Matthew Caesar Jennifer Rexford
Princeton University Princeton University Princeton University

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 SEIZE that achieves the best of both worlds: The scalability of IP combined with the simplicity of Ethernet. SEIZE provides plug-and-play functionality via flat addressing, while ensuring scalability and efficiency through shortest-path routing and hash-based location resolution. We implemented a prototype of SEIZE using the Click and XORP open-source routing platforms, and evaluated system performance on Emulab. Additionally, to evaluate performance on larger scales, we performed a simulation study driven by real-world traffic traces and network topologies. Our experiments show that SEIZE attains near-optimal path efficiency, while reducing control overhead and table size by roughly two orders of magnitude compared with Ethernet bridging.

1. Introduction

Ethernet, developed in the mid 1970's, stands as one of the oldest networking technologies still in use today. It has weathered the years extremely well due to its simplicity and ease of configuration, and has become a widely used building-block in many enterprise and access provider networks. Each host in an Ethernet is assigned a persistent MAC address, and switches automatically learn host addresses and locations. The "plug-and-play" semantics simplify many aspects of network configuration. Since addressing is persistent, many access controls may be statically configured. Flat addressing simplifies the handling of topology changes and host mobility, without requiring administrators to perform address reassignment.

Despite these benefits, Ethernet is rarely used outside of small Local Area Networks (LANs), for three main reasons [1]. First, Ethernet relies on network-wide flooding to discover the location of end hosts, resulting in large state requirements and control message overhead that grow with the size of the network. Scalability is a growing concern due to increasing levels of host mobility, as well as network deployments in developing regions where the topology may change frequently [2]. Second, Ethernet does not allow arbitrary path selection, but forces paths to comprise a *spanning tree*. Spanning trees perform well for small networks which often do not have much physical redundancy anyway, but introduces substantial inefficiencies on larger networks that have more demanding requirements for low latency, load

balancing, and traffic engineering. Finally, popular bootstrapping protocols, such as ARP (Address Resolution Protocol) and DHCP (Dynamic Host Configuration Protocol), rely on broadcasting, which consumes excessive resources and introduces security vulnerabilities and privacy concerns.

Network administrators sidestep these problems today by interconnecting small Ethernet-based LANs using IP routers. IP ensures efficient and flexible use of networking resources via shortest-path routing, and has control overhead and forwarding-table sizes that depend on the number of IP prefixes, rather than the number of hosts. However, introducing IP routers breaks many of the desirable properties of Ethernet. For example, network administrators must now subdivide their address space and assign IP prefixes across the topology, leading to wasted address space and complex configuration tasks. Although DHCP automates host address configuration, maintaining consistency between DHCP servers and routers remains challenging. Meanwhile, administrators must reconfigure their DHCP servers and routing protocols as the network design changes. Moreover, since IP addresses are not persistent identifiers, access-control policies must be specified based on the host's current position, and updated when the host moves.

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 this question, we present a scalable and efficient, zero-configuration enterprise (SEIZE) architecture. Specifically, SEIZE offers the following key features:

- **Routing on persistent identifiers:** To enable plug-and-play routing, SEIZE automatically discovers end-host MAC addresses and locations, and forwards packets based on MAC addresses. This minimizes configuration and simplifies handling of mobile hosts.
- **Efficient use of paths:** For efficient resource utilization and intuitive traffic engineering, SEIZE delivers packets along shortest paths. A link-state routing protocol maintains a map of the switch-level interconnection topology, without disseminating the end-host location information, similar to the way routing is done inside Internet Service Provider (ISP) networks.
- **Reducing control overhead and forwarding state:** Despite relying on per-host information to forward packets, SEIZE does *not* require each switch to discover or maintain state for all end hosts, nor does

it require network-wide floods to discover host location. Instead, SEIZE adopts a *hash-based, on-demand* location-resolution scheme indexed on hosts' identifiers. In particular, SEIZE leverages the network-wide view provided by a link-state routing protocol to form a one-hop DHT [3], which stores the network location of each host. Switches cache host information to optimize the packet-forwarding paths and to avoid unnecessary queries. In enterprise networks, hosts typically communicate with a small number of other hosts [4], making on-demand resolution and caching quite effective.

Despite these novel features, SEIZE remains backwards-compatible with existing applications and protocols running at end hosts. For example, SEIZE allows hosts to generate broadcast ARP and DHCP messages, and internally converts them into unicast-based queries to a directory service. SEIZE switches can also handle general (i.e., non-ARP and non-DHCP) broadcast traffic through multicasting. To offer broadcast scoping and access control, SEIZE introduces a flexible grouping scheme that is compatible with Virtual LANs (VLANs).

This paper makes three key contributions. First, we provide a very simple, and yet highly scalable mechanism that enables shortest-path forwarding while maintaining the semantics of Ethernet (Sections 3 and 4). Second, we leverage simulations to evaluate the protocol over a wide variety of workloads and network topologies, and analyze performance (Section 5). Third, we describe a prototype implementation of SEIZE, and evaluate its performance through a combination of microbenchmarks and end-to-end measurements (Sections 6 and 7). Our initial results are promising: SEIZE scales to networks containing two orders of magnitude more hosts than a traditional Ethernet. The first few packets of a flow are subject to a small delay penalty, while the remaining packets traverse shortest paths. SEIZE can also handle network failures 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 SEIZE, this section explains why Ethernet bridging is limited to small LANs. Then we explain how hybrid IP/Ethernet designs improve some aspects of scalability, but introduce management complexity, eliminating many of the "plug-and-play" advantages of Ethernet.

2.1 Ethernet bridging

An Ethernet network is composed of *segments*, each comprising a single physical layer¹. Ethernet *bridges* are used to connect multiple segments into a multi-hop network, namely a LAN, forming a single broadcast domain. Since too large a broadcast domain unnecessarily overloads bridges and end

¹In modern switched Ethernet networks, a segment is just a point-to-point link connecting an end host and a bridge (switch), or a pair of bridges. Thus, we use *segment* and *link* interchangeably.

hosts, administrators often *logically* subdivide the domain into several smaller broadcast domains called VLANs. A VLAN allows a set of hosts on different segments to communicate as though they were attached to the same physical wire. Each host in an Ethernet is assigned a globally unique 48-bit MAC (Media Access Control) address. A bridge learns how to reach hosts by inspecting the incoming frames (packets), 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 does not appear in the forwarding table, the bridge sends the frame on all outgoing ports, initiating a network-wide flood. In addition to ensuring the frame reaches its destination, flooding allows the rest of the network to learn the source's location. Bridges also flood frames that are destined to a broadcast MAC address.

Despite Ethernet's reliance on flooding and broadcasting, bridges cannot detect forwarding loops because the Ethernet frame does not carry a TTL value. Thus, networks with rich topologies can experience *broadcast storms*, where frames are repeatedly replicated and forwarded. To avoid this, bridges coordinate to compute a *spanning tree* that is used to forward frames. Network administrators first select and configure a single *root bridge*; then, the bridges collectively compute a spanning tree. Links not present in the spanning tree cannot be used to carry traffic, leading to longer paths and inefficient use of resources. The inefficiency of spanning trees and the reliance on network-wide flooding prevents Ethernet from being adopted as a general interconnection mechanism for large networks.

Globally disseminating every host's location: Flooding and source-learning introduce two problems in large networks. First, the forwarding table at each bridge can grow very large because the table size remains proportional to the number of end hosts in the network. A large forwarding table makes bridge design more challenging and expensive. To boost look-up speed, most Ethernet bridges use CAM (Content-Addressable Memory) to implement a forwarding table. This type of memory is quite expensive and power intensive, so using a smaller-sized CAM is important. This requirement is particularly critical in building low-end bridges, such as wireless access points. Second, the control overhead required to disseminate each end host's information can be very large, wasting link bandwidth and processing resources. Since hosts power up/down and change location relatively frequently compared to network infrastructure changes, 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, address scanning attacks [5].

Inflexible route selection: Larger enterprise networks often have richer topologies, for greater reliability and performance. Forcing all traffic to traverse a single spanning tree leads to suboptimal paths and uneven link loads, with especially high load on the links near the root bridge. Choosing

the right root bridge is extremely important, imposing an additional burden on network administrators; still, even with a good choice for the root bridge, a spanning tree is not an efficient way to forward traffic. As a workaround, administrators may configure multiple VLANs, and use a separate, disjoint spanning tree for each VLAN [6, 7]. Although an improvement over using a single spanning tree, effective use of per-VLAN trees requires fine-tuning each tree's path, which must be manually updated as traffic shifts. Moreover, traffic within each VLAN must traverse a spanning tree, resulting in suboptimal performance.

Dependence on broadcasting for basic operations: DHCP and ARP are essential protocols 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 typically generated more frequently, whenever a host needs to know the IP address of another host in the same broadcast domain. Since hosts generate ARP queries individually, the overhead grows in proportion to the number of communicating host pairs. Back in the era of shared-medium Ethernet, broadcasting DHCP and ARP queries was a reasonable design choice. However, now that most Ethernet networks consist of point-to-point links, relying on broadcasting for these frequent operations is extremely wasteful [8]. In addition to degrading network performance, every broadcast message must be processed by every end host; since handling of broadcast frames is often application or OS-specific, these frames are typically *not* handled by the network interface card, and instead must interrupt the CPU [9]. Additionally, 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 network-wide ARP or DHCP floods [8].

2.2 Hybrid IP/Ethernet architecture

To deal with Ethernet's shortcomings in larger-scale environments, administrators typically build enterprise and access provider networks out of multiple Ethernet LANs interconnected by *IP routing*. In these *hybrid* networks, each LAN corresponds to an *IP subnet* and is given an *IP prefix* representing the subnet. Each host in a LAN is 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 hierarchical topology, yet must avoid fragmentation and wasted namespace, and must take into account future use of addresses to minimize later reassignment. Unlike a MAC address, which functions as a host *identifier*, an IP address functions as a *locator*, which denotes the host's current location in the network. Since forwarding a packet involves both the MAC and IP addresses, routers run protocols such as ARP to map between the two.

IP routers in a hybrid network cooperate to compute paths

among themselves using a *routing protocol* such as EIGRP (Enhanced Interior Gateway Routing Protocol) [10] or OSPF (Open Shortest Path First) [11]. Since the IP routing protocol uses a different addressing model and path-selection process from Ethernet, designing mechanisms to share routing information across the two protocols has been a challenging problem. Instead, typical deployments run the two protocols independently and connect them together at certain fixed locations called *default gateways*. Unfortunately, this prevents the shortest network-level paths from being visible across LANs, making traffic engineering challenging for network administrators to implement.

Nevertheless, 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 [12]. 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 the end-host configuration, but introduces substantial configuration overhead for managing the DHCP servers. In addition, maintaining consistency between subnet router configuration and DHCP address allocation configuration is not a simple matter. Worse yet, maintaining consistency among distributed DHCP servers is particularly challenging because there is no standard coordination protocol that can automatically adjust address allocation policies across multiple DHCP servers. Finally, even after this configuration has taken place, network administrators must continually revise it to handle network design changes.

Complexity in implementing networking policies:

Administrators today use a collection of access controls, QoS (Quality of Service) controls [13], and other policies to control the way packets flow through their networks. Today these policies are typically defined based on IP prefixes. However, since IP address ranges are assigned based on the topology, changes to the network design require these policies to be rewritten. For the same reason, when a host's IP address changes due to mobility, a policy defined on the host's old address must be revised. One way to deal with this is to configure one or several network-wide VLANs to allow hosts to retain their IP address regardless of location. However, all hosts on the same VLAN must be within the same IP subnet in order to avoid all intra-LAN routes going through the default gateway. Hence, deploying large VLANs increases the total amount of broadcast traffic in the network and VLAN state in switches as well. A second problem IP introduces is that rewriting networking policies must happen immediately after the network design changes to prevent

reachability problems and to avoid vulnerabilities. All these problems stem from the fact that IP addresses are not assigned in a persistent fashion. Ideally, administrators should only need to update policy configurations when the *policy* itself changes, not when the *network* changes.

3. Scaling Ethernet With SEIZE

In this section, we describe the mechanisms SEIZE uses to scale to large networks, deferring discussion of performance enhancements to Section 4. We first introduce our main design decisions: flat addressing and shortest-path forwarding. Then we describe how SEIZE manages host-location information in a scalable fashion. Finally, we explain how SEIZE responds to network changes, such as switch failures and host mobility.

3.1 Flat addresses & shortest-path forwarding

SEIZE does not require changes to the addressing structure currently deployed in Ethernet networks. Packet forwarding is based on destination MAC addresses. Paths used for packet delivery are computed based on a switch-level topology maintained by a link-state routing protocol.

3.1.1 MAC-based addressing and forwarding

Each end host in a SEIZE network is identified by a *MAC address*, a 48-bit permanent identifier, and switches deliver packets based on their destination MAC addresses. Since MAC addresses are globally unique and hard-coded in each Ethernet interface card, end-host configuration is not necessary. This ensures backwards compatibility with the existing Ethernet interface cards, device drivers, and protocol stacks.

End hosts may need IP addresses for application-level compatibility and external reachability, but not for routing within a SEIZE network. As such, the use of SEIZE simplifies IP address assignment, since a single large address block suffices for all hosts in the network. An end host can use any unique IP address from the large prefix pool, regardless of its current location. To identify the MAC address associated with an IP address, end hosts use conventional ARP. To prevent broadcast ARP requests from overwhelming the entire network, SEIZE handles these requests via a more scalable mechanism, as discussed later in Section 4.2.

3.1.2 Link-state protocol to maintain switch topology

Shortest-path forwarding requires that every SEIZE switch knows the current network topology. SEIZE enables this by running a link-state protocol. However, distributing *end-host* information in link-state advertisements (LSAs), as advocated in previous proposals [8, 14], would lead to serious scaling problems. Instead, SEIZE’s link-state protocol maintains only the *switch-level* topology, which is much more compact and stable. SEIZE 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, SEIZE switches run a discovery protocol to determine which of their links are attached to hosts, and which are attached

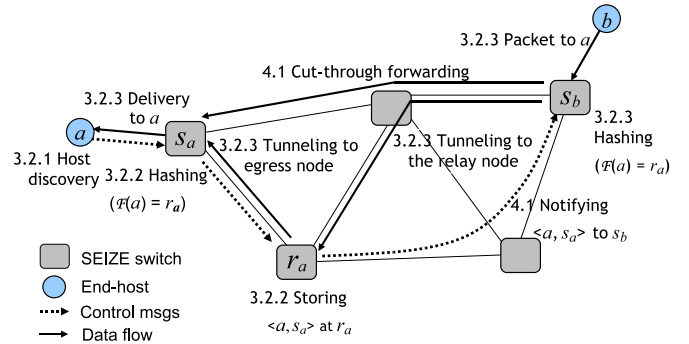


Figure 1: Packet forwarding and lookup in SEIZE.

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, SEIZE switches determine their own unique *switch IDs* without administrator involvement. Each switch does this by choosing the MAC address of one of its interfaces as its switch ID.

3.2 Managing host location information

Since per-host information is not disseminated in link-state advertisements, SEIZE switches need some other way to determine a destination host’s location. This is done in an on-demand fashion, via a simple hashing mechanism. In particular, each host willing to receive packets uses consistent hashing to map its MAC address to a switch identifier, and places a *pointer* at that switch to the host’s current location. We refer to this switch that stores the location as the *relay*. A remote host may then contact the destination by sending the packet via its access switch, which uses the same hash function to identify which switch is the relay for the destination. The access switch then forwards the packet to the relay, which in turn uses the pointer to forward the packet to the destination. Figure 1 illustrates each step of this procedure, annotated with the subsection number in which the step is described.

3.2.1 Host discovery by access switches

When an end host arrives at a SEIZE network, its access switch discovers the host using traditional Ethernet mechanisms. The access switch is responsible for forwarding packets to and from the end host, and ensuring the end host’s relay correctly maintains the end host’s location. Similarly, when an end host fails or disconnects from the network, this access switch is responsible for detecting that the host has left the network, and revoking the end host’s location information from the network.

Switches can detect a host’s arrival and departure either explicitly or implicitly, depending on the underlying link-layer technology. For example, if the host’s Ethernet segment runs an 802.11-style network-association protocol, the access switch can explicitly detect hosts’ arrival and departure. When such options are not available, switches can use

the conventional 802.1D MAC learning scheme to detect arrivals. Unfortunately, detecting departure in this case requires polling end hosts periodically or monitoring loss of link-layer acknowledgment messages.

3.2.2 Location registration

To efficiently maintain each end host’s location information, all SEIZE switches jointly use a hash function \mathcal{F} . \mathcal{F} receives a MAC address as an input, maps the input to one of the switches in the network, and returns the mapped switch’s identifier. Since each switch knows all the other switches’ identifiers via link-state advertisements from the routing protocol, \mathcal{F} works identically across all switches.

When host a first arrives at access switch s_a , the switch learns a ’s id mac_a , and computes $\mathcal{F}(mac_a)$. This hash value is another switch r_a ’s id: $r_a = \mathcal{F}(mac_a)$. Finally, s_a informs r_a of a ’s id mac_a and a ’s location s_a . Switch r_a , the *relay switch* for host a , then keeps a ’s location information in the form of the tuple (a, s_a) . If desired, standard DHT load balancing mechanisms may be used to deal with switch heterogeneity [15].

3.2.3 Location resolution and packet forwarding

Suppose host b connected to a different switch s_b wants to communicate with a . Since the link-state protocol does not disseminate end-host information, when s_b receives a packet destined to a , it does not know how to reach a . Unlike Ethernet bridging, where s_b would flood the packet, instead the SEIZE switch s_b computes $\mathcal{F}(mac_a) = r_a$ and sends the packet to r_a via unicast. Since r_a might be multiple hops away from s_b , and the intermediate nodes between s_b and r_a might not know the location of s_a , s_b encapsulates the original packet inside an additional header. This outer header uses r_a as a destination address and uses link-state routing to travel from s_b to r_a . Upon receiving the packet, r_a decapsulates it, looks up the destination address mac_a in its forwarding table, and obtains s_a . Switch r_a then re-encapsulates the packet and sends it to s_a . Packets are forwarded along the shortest paths between s_b and r_a , and also between r_a and s_a .

3.3 Responding to changes

There are two kinds of changes that can occur in a network. First, 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. Second, the host-location information may change, if a new host joins the network, or an existing host fails or moves to a new switch.

3.3.1 Handling network topology changes

In networks, links or switches may fail, and the failure 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/repair that does not partition a network, the set of switches appearing in the link-state map

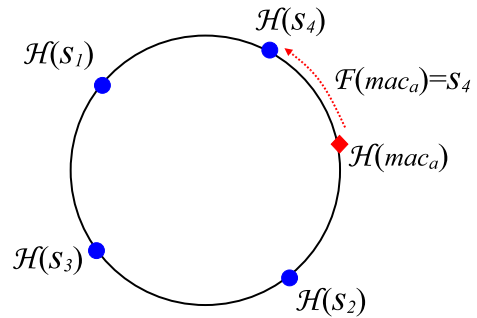


Figure 2: Host addresses are consistently hashed onto switches (s_i).

does not change. Since the hash function \mathcal{F} is defined with the set of switches in the network, the hash function \mathcal{F}^{new} after the topology change is equivalent to the hash function \mathcal{F}^{old} before the change. Thus, for every host h , its old relay switch $r_h^{old} = \mathcal{F}^{old}(mac_h)$ is identical to its new relay switch $r_h^{new} = \mathcal{F}^{new}(mac_h)$ after the 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 SEIZE, this is simply handled by the link-state routing protocol.

However, if a switch fails or recovers, the set of switches in the link-state map changes. Since there are now fewer or more switches, there may be some hosts h whose r_h^{old} differs from r_h^{new} . To deal with this, two operations must take place. First, for correctness reasons a host h ’s location information must be inserted at h ’s new relay switch r_h^{new} , since remote switches will begin forwarding packets destined for h via r_h^{new} . SEIZE accomplishes this by having h ’s access switch monitor h ’s current relay’s liveness by observing link-state advertisements. When the current relay fails, s_h computes $\mathcal{F}^{new}(mac_h)$, obtains a new relay switch r_h^{new} and re-registers h ’s location information with the new relay. Second, h ’s location information must eventually be removed from r_h^{old} for garbage collection purposes, since remote switches will no longer be forwarding packets via r_h^{old} to h . SEIZE relays accomplish this by triggering a scan over stored location information when the link-state topology changes. Entries that no longer hash to the local switch are evicted. This eviction takes place after a grace period to ensure correct packet delivery during the link-state convergence.

This procedure correctly handles network partitions. The link-state protocol ensures that each switch will be able to see only switches present in its partition. Therefore, each network partition comes to use the same hash function applied over a different set of switches. If two hosts are part of the same partition, SEIZE will correctly forward packets between them. When a host sends a packet to a host belonging to an unreachable partition, the packet will be discarded by the relay switch, as the switch will not be aware of the destination host’s location.

To minimize the amount of control overhead required to deal with topology changes, SEIZE utilizes *Consistent Hashing* [16] for \mathcal{F} . This mechanism is illustrated in Figure 2. A consistent hashing function maps keys to *bins* such that the arrival or departure of a bin causes minimal changes

in the mapping of other keys to bins. In SEIZE, each switch comprises a bin, and each host MAC address corresponds to a key. Formally, given a set $S = \{s_1, s_2, \dots, s_n\}$ of switch identifiers, and an end host a and its MAC address mac_a ,

$$\mathcal{F}(mac_a) = \underset{s_i \in S}{\operatorname{argmin}} \{\mathcal{D}(\mathcal{H}(mac_a), \mathcal{H}(s_i))\}$$

where \mathcal{H} is a regular hash function², 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 host to the switch with the closest identifier not exceeding that of the host on the hash space of \mathcal{H} . As an optimization, a host may be additionally mapped to the next k 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.

The use of consistent hashing ensures that on average, a single switch failure or recovery triggers $|N|/|S|$ host re-registration events, where $|N|$ is the number of end hosts and $|S|$ is the number of switches in the network. That is, in a network with 100,000 end hosts and 1,000 switches, a single switch failure generates only 100 host re-registration messages on average.

3.3.2 Handling host location changes

When a host h moves to a new location, three things must happen. First, h 's new access switch s_h^{new} must learn that it is responsible for the host. Second, h 's relay switch r_h must learn of h 's new location. The mechanisms introduced in Section 3.2.1 and Section 3.2.2 handle this by having s_h^{new} send a registration message to r_h upon arrival of h . Note that the relay switch r_h selected by \mathcal{F} does not change when h 's location changes. Finally, h 's old access switch s_h^{old} must learn that it is no longer responsible for h . SEIZE handles this by having s_h^{old} deregister h if it detects h is unreachable. As an optimization, the relay switch r_h notifies s_h^{old} with a pointer to the new location s_h^{new} before r_h updates h 's location information.

4. SEIZE Performance Enhancements

The previous section left several performance issues unaddressed. First, forwarding every data packet through the relay would result in longer paths, with higher delay. In this section we describe an optimization based on caching that allows the majority of packets to traverse the shortest path to the destination. Second, in order to support legacy Ethernet hosts, SEIZE must support broadcast traffic. To deal with this, we describe how switches handle ARP and DHCP through unicast-based queries, and how administrators can scope broadcast traffic. Finally, we describe several other performance-related optimizations, including latency reduction in a geographically distributed network. Note that, with all these mechanisms for performance enhancements and backwards-compatibility, SEIZE still preserves its key principle: to make host information available only when and where it is needed.

²In our prototype, we used MD5 as \mathcal{H} .

4.1 Reducing path inflation

By storing each host's state only at its access and relay switches, SEIZE reduces control overhead and forwarding state. However, by delivering all packets through a relay switch, path lengths may potentially increase in length up to twice the network's diameter. In this section we describe an optimization called *cut-through forwarding*, which is based on a similar mechanism used in some ATM (Asynchronous Transfer Mode) networks [17]. Cut-through forwarding allows most packets of a flow to traverse the shortest path to the destination. Additionally, we describe how administrators can build a hierarchical SEIZE network to further reduce path inflation.

4.1.1 Cut-through forwarding (CTF)

CTF works by having the relay switch inform the ingress switch of the destination's location. The ingress switch then temporarily caches this information, and uses it to forward packets directly along the shortest network-level path without traversing the relay switch. This mechanism is illustrated in Figure 1. When host a 's relay switch r_a receives a packet destined to a from s_b , r_a recognizes that this is a relayed packet because a is not directly connected to r_a . Thus r_a notifies s_b that a 's current location is s_a . Then s_b temporarily caches this information in its forwarding table and uses it to deliver subsequent packets directly to s_a . Although CTF requires additional control messages to inform the ingress of the host's location, its benefit is much larger than its penalty especially in enterprise and access provider networks because most hosts communicate with a small number of popular hosts, such as mail/file/Web servers, printers, VoIP gateways, and the Internet gateway [4]. Additionally, to prevent forwarding tables from growing unnecessarily large, the ingress switch s_b applies various cache-management policies. For correctness, however, the cache-management scheme must not evict the location information of the hosts that are directly connected to s_b or are registered with s_b for relaying.

Use of CTF requires a slight modification to the host mobility handling mechanism introduced in Section 3.3.2 because entries cached by CTF must be kept up-to-date as hosts move. If CTF is disabled, when host a moves from s_a^{old} to s_a^{new} , it is sufficient for s_a^{new} to update a 's old location information at the relay switch r_a , and for r_a to update the old access switch s_a^{old} . When using CTF, even after updating the information at r_a , s_a^{old} may receive packets destined to a because other switches in the network might have stale information in their forwarding tables. To deal with this, switches discard cached entries after a short timeout period. In addition, when s_a^{old} receives packets destined to a , it can explicitly notify ingress switches that sent the packets of a 's new location. To minimize service disruption, s_a^{old} also forwards those misdelivered packets to s_a^{new} .

4.1.2 Localizing paths with hierarchy

When using SEIZE in a large, geographically distributed network (e.g., an enterprise composed of multiple remote

sites interconnected by VPNs), relaying could result in a large latency penalty, as the relay may lie far from the source and destination. Moreover, forwarding through a large number of switches that are not on the shortest path may degrade overall communication quality (i.e., loss probability, availability, etc.). These problems become even worse when source and destination are very close to each other. Although CTF ensures that only the first few packets of each flow experience these problems, some special applications that are highly sensitive to path quality can still suffer.

To deal with this, SEIZE may be configured in a hierarchical fashion. A hierarchical network is divided into several *regions*, and a *backbone* providing long-distance connectivity across regions. Each region is connected to the backbone via its own *border switch*, and the backbone is composed of the border switches of all regions. Every 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, SEIZE ensures that regional traffic is handled entirely within its own region, and only inter-region traffic is forwarded through the backbone. SEIZE ensures this by defining a separate *regional* and *backbone* hash ring. When a host a joins a region P and is registered with a regional relay r_a^P (i.e., a relay switch for a in region P), r_a^P additionally forwards a ’s information to region P ’s border switch b^P . Then b^P hashes a ’s MAC address again onto the backbone ring and registers a to another backbone switch b_a^Q . Switch b_a^Q may then cache a ’s location. As an optimization to reduce load on border switches, b_a^Q may hash a and stores a ’s location at a switch within its own region Q , rather than storing a ’s location directly. A similar process is used to forward packets from a remote host to a .

4.2 Reducing broadcast overhead

To perform the same semantics as Ethernet bridging, SEIZE needs to support broadcasting. However, in a large network, broadcasting may significantly overload switches and end hosts. SEIZE presents two independent mechanisms that both contribute to solving the problem. First, we propose to sidestep the problem of broadcasting by converting ARP and DHCP queries to unicast messages. Second, to handle other broadcast traffic efficiently, we propose a *scoping* mechanism which is similar to, but more flexible than, VLANs.

4.2.1 Eliminating broadcasting from ARP and DHCP

To support ARP without network-wide broadcasting, SEIZE switches use the same hash-based resolution scheme introduced in the previous section but with *IP addresses* as keys. Upon learning host a ’s IP address, the access switch s_a^3 , s_a registers a ’s MAC address mac_a and IP address ip_a with a *resolver switch* $\mathcal{F}(ip_a)$. Later, when another host b issues a broadcast ARP request to resolve a ’s MAC address

³This could happen when an explicit association process between a and s_a takes place. When explicit association is not available, s_a can still learn this from the first IP packet sent by a or from DHCP messages sent to a .

mac_a associated with ip_a , b ’s access switch s_b converts the request to a unicast query to $\mathcal{F}(ip_a)$, instead of broadcasting the request network-wide.

SEIZE also 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 network-wide. SEIZE implements this mechanism using the existing DHCP relay agent standard [18]. This standard is used when an end host needs to directly communicate with a DHCP server that is located in a different broadcast domain. The standard proposes that a host’s IP gateway forward a DHCP discovery to a DHCP server via IP routing. In SEIZE, a host’s access switch can perform the same proxy function with Ethernet encapsulation.

4.2.2 Group-based broadcast scoping

In addition to ARP and DHCP, there are several other sources of broadcast traffic, including IP multicast, service discovery, and file sharing programs. To support this broadcast traffic in a scalable fashion, we need a VLAN-like broadcast scoping mechanism. As a solution, SEIZE introduces a notion of *group*. A group is defined as a set of hosts who share the same broadcast domain regardless of their locations. Unlike VLANs, however, a broadcast domain in SEIZE does not limit unicast reachability between hosts because a SEIZE switch can resolve any host’s location in the network without relying on domain-wide flooding. Thus, groups provide several additional advantages over Ethernet’s VLAN-based approach. First, a group does not have to correspond to an IP subnet. Each host in a group can use its own site-local IP address, and changing groups does not force a host to change its IP address. Second, a unicast flow between two hosts belonging to different groups takes the shortest path, not through IP gateways.

With this definition of a group, broadcast scoping works as follows. 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 again determined by the same hash function \mathcal{F} using a group’s identifier as an input. When a switch, for the first time, detects an end host that is a member of group g^4 , the switch issues a join message that is carried up to the nearest graft point on the tree toward g ’s broadcast root. Note that, by virtue of \mathcal{F} , all the switches along the new branch taken by the join message can consistently map group g to its broadcast root. When a host departs, its access switch prunes a branch if necessary. As a result, each switch in a network maintains local knowledge about which of its interfaces belongs to which groups. Finally, when an end host in g sends a broadcast frame, the host’s access switch encapsulates and forwards the frame

⁴The way administrators associate hosts with corresponding groups and policies is beyond the scope of this paper. For Ethernet, a policy management framework that can automate this task (e.g., mapping an end host or flow to a VLAN) is already available [19], so SEIZE can employ the same model.

along g 's multicast tree. In a manner similar to IP multicast's RPF (Reverse Path Forwarding) mechanism, each switch ignores packets received from the interface that points to the source, thereby ensuring loop-freedom and avoiding broadcast storms.

4.2.3 Separating access control from broadcasting

Unlike Ethernet bridging, which unavoidably misuses VLANs for both broadcast scoping and access control, SEIZE can separate the two functions because groups scope only broadcast traffic. For unicast scoping (i.e., access control), groups in SEIZE only provide a name space upon which access-control policies are defined. Specifically, the access-control mechanism in SEIZE works as follows. First, each host's group membership is determined by the host's access switch and registered with the host's relay switch along with other information (e.g., location, or addresses). Access control policies are then applied when a host attempts to resolve a destination's location. In particular, when a source host b sends a unicast packet to a , b 's access switch forwards the packet to the relay r_a for a . The relay switch forwards the packet to a only if the access-control policy registered at r_a permits communication between b 's group g_b and a 's group g_a .

5. Simulations

In this section, we start by describing our simulation environment. Next, we describe SEIZE's performance under workloads collected from several real operational networks. We also investigate SEIZE's performance in alternate environments by generating host mobility and topology changes. In Section 7 we will also describe some validation to cross-check our simulation results with those of the prototype implementation.

5.1 Methodology

To evaluate performance of SEIZE, we would ideally like to have several pieces of information: 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 administrators typically do not log events on such large scales. To deal with this, we leveraged real traces where possible, and supplemented them with synthetic traces. To generate the synthetic traces, we made several assumptions about workload characteristics, and varied these characteristics to measure the sensitivity of SEIZE to our assumptions.

In our simulator, we replayed packet-level traces collected from the LBNL (Lawrence Berkeley National Lab) campus network by Pang et. al. [20]. 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. To evaluate sensitivity of SEIZE to net-

work size, we artificially injected additional hosts into the trace. We did this by creating a set of virtual hosts, each of which communicated with a set of random destinations. We selected this set so as to ensure that the distribution of incoming and outbound flows across hosts was not substantially changed. We also tried injecting MAC scanning attacks and artificially increasing the rate at which hosts send [5].

We measured SEIZE's performance on four representative topologies. *Entp-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 [21]. Because SEIZE switches are intended to replace both IP routers and Ethernet bridges, the routers in these topologies are considered as SEIZE switches in our evaluation. To investigate a wider range of environments, we also constructed a model topology called *Entp-model*, which represents a typical enterprise network composed of four full-meshed core switches each of which connects to a tree of twenty one switches. This roughly characterizes a commonly-used topology in building enterprise networks [22].

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 [23] to roughly characterize this number for networks reachable from the Internet. However, the CAIDA skitter traces form a sample representative of the wide-area, it is not clear they apply to the smaller-scale networks we model. Hence for *Entp-model* and *Entp-campus*, we assume that hosts are evenly distributed across switches.

Given a fixed topology, the performance of SEIZE and Ethernet bridging can vary depending on input traffic patterns. To quantify this variation we repeated each simulation run 25 times, and plot the average of these runs with confidence intervals. For each run we vary the seed to the simulator, which causes the number of hosts per switch, and the mapping between hosts and switches, to change. Additionally for Ethernet bridging's case, we varied spanning trees by randomly selecting one of the core switches as a root bridge.

5.2 Baseline performance

5.2.1 Stretch

Figure 3a shows the relative latency penalty, or *stretch*, of Ethernet and SEIZE within the *Entp-model* topology. We measure stretch by dividing the time the packet was in transit by the delay along the shortest path through the topology. Ethernet incurs some small stretch because it sends packets over a spanning tree. We compute Ethernet's spanning tree by computing the shortest path from the root bridge to each switch, using link delay as a metric. When CTF is disabled (*SEIZE_REL*), stretch is increased by an additional amount. However, when CTF is enabled (*SEIZE_CTF*), the stretch penalty nearly disappears. This is because only the first few packets of a flow must traverse the relay, while the others

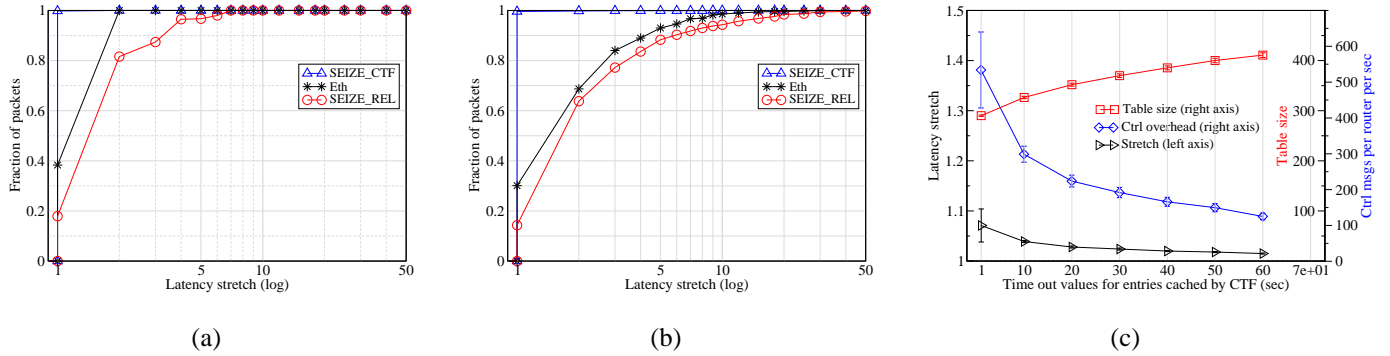


Figure 3: (a) CDF of stretch in *Entp-model* (b) CDF of stretch in *Ap-large* (c) Effect of CTF cache timeout in *Ap-large*.

traverse the shortest path. We identified similar results on the *Entp-campus* topology, which is another local network.

Figure 3b shows stretch on the AS 1239 topology (*Ap-large*). Interestingly, stretch increases as compared to the smaller *Entp-model* topology. This happens because AS 1239 has a larger number of long-distance links, so forwarding over non-shortest paths via Ethernet’s spanning tree incurs larger costs. Unfortunately, *SEIZE_REL* increases this penalty even further. However, using CTF reduces this penalty to 1 for nearly all packets. We found *SEIZE* performs similarly on the AS 3967 topology (*Ap-small*). If desired, the path localization scheme introduced in Section 4.1.2 could be applied to further reduce stretch.

In our experiments, we used CTF with a timeout-based cache eviction policy with entries evicted after 60 seconds of unused period. To verify the effect of the timeout setting on stretch, we varied it from 1 to 60 seconds. As shown in Figure 3c, CTF performs well across a wide range of timeout values.

5.2.2 Forwarding table size

Figure 4a shows the amount of state per switch in the *Entp-large* topology. Ethernet requires more state than *SEIZE* with CTF disabled (*SEIZE_REL*). This happens because Ethernet stores per-host information entries at almost every bridge. In a network with s switches and h hosts, each Ethernet bridge must store an entry for each destination, resulting in $O(sh)$ state across the network. However *SEIZE* requires $O(h)$ state since only the access and relay switches need to store location information for each destination. In this particular topology, *SEIZE* reduces forwarding-table size by roughly a factor of 22. As shown in Figure 4b, these gains increase to a factor of 64 in *Ap-large* because there are a larger number of switches in that topology. Unfortunately, while the use of CTF drastically reduces stretch, we can see that it increases *SEIZE*’s average forwarding-table size by roughly a factor of 1.5. However, even with this penalty, *SEIZE* 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 cut-through forwarding entries, so as to trade off stretch and space requirements. Finally, Figure 3c shows that the increase of

table size due to a larger CTF timeout is modest.

5.2.3 Control overhead

Figure 4c shows the amount of control overhead generated by *SEIZE* and Ethernet. We compute this value by computing the total number of control messages — unnecessarily flooded packets in the case of Ethernet bridging — over all links in the topology during the experiment, then dividing by the number of switches, then dividing by the duration of the trace. *SEIZE* significantly reduces control overhead as compared to Ethernet. This happens because Ethernet generates network-wide floods for the significant majority of packets, while *SEIZE* leverages unicasts to disseminate host location. Here we again observe that use of CTF degrades performance slightly. Namely, *SEIZE_CTF* increases control overhead roughly from 0.1 to 1 packet per second as compared to *SEIZE_REL* in a network containing 30K hosts. However, *SEIZE_CTF*’s overhead still remains a factor of 1000 less than that of Ethernet. As shown in Figure 3c, *SEIZE_CTF* retains low control overhead as long as the CTF cache timeout is not set unreasonably small.

5.3 Sensitivity to network dynamics

5.3.1 Effect of network instability

Figure 5a shows performance during switch failures. Here, we cause switches to fail randomly, with failure interarrival times drawn from a Pareto distribution with $\alpha = 2.0$, with varying mean values. Switch recovery interarrival time is also drawn from the same distribution, with an average of 30 seconds. We found *SEIZE* is able to deliver a larger fraction of packets than Ethernet. This happens because *SEIZE* 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. We also found that use of CTF improved availability. This is because traffic forwarded through a relay shares fate with a larger number of switches, including the relay itself.

5.3.2 Effect of host mobility

To investigate the effect of host mobility on *SEIZE* performance, we randomly move hosts between access switches.

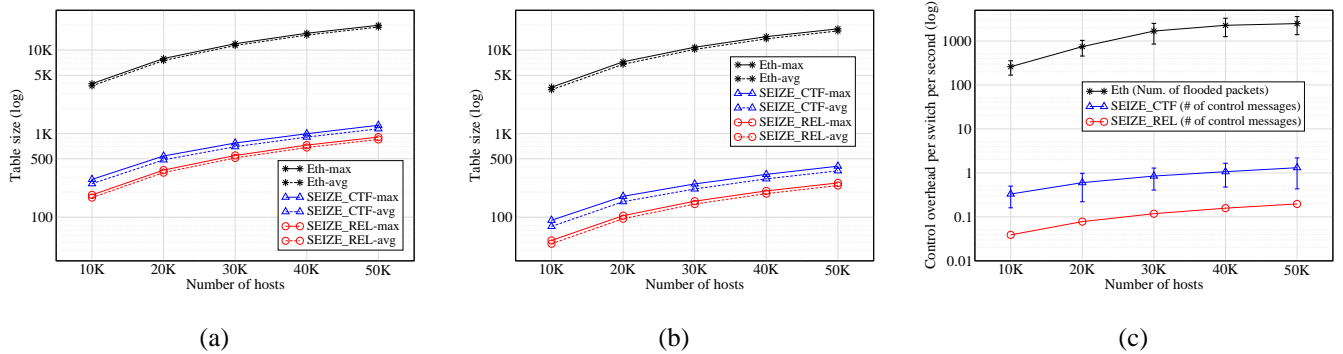


Figure 4: (a) Table size increase in *Entp-model* (b) Table size increase in *Ap-large* (c) Control overhead in *Ap-large*.

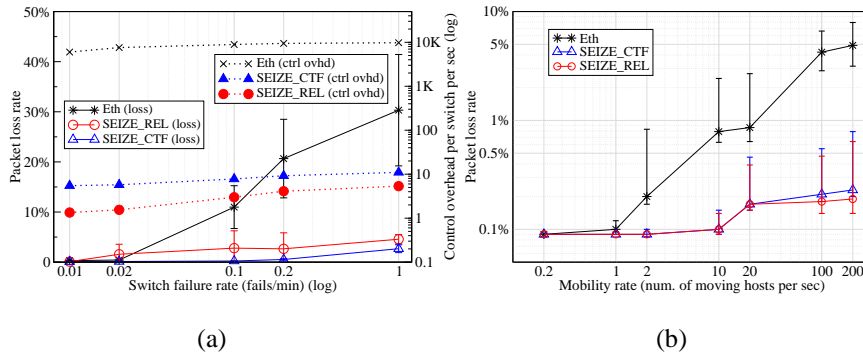


Figure 5: Sensitivity to network dynamics: Effect of (a) switch failure in *Entp-model* (b) mobility in *Entp-campus*.

We drew mobility times from a Pareto distribution with $\alpha = 2.0$ and varying interarrival times. For high mobility rates, we found SEIZE’s loss rate was lower than that of Ethernet, as shown in Figure 5b. This happens because when a host moves in an Ethernet, it takes some time for switches to evict the stale location information, and re-learn the host’s new location. Although some Ethernet implementations flood when a host moves, this increases control overhead. On the other hand, SEIZE provides both low loss and control overhead by relying on unicasts to update host state.

6. Implementation

Our simulation results indicate that SEIZE performs efficiently on several network topologies. To verify SEIZE’s performance and practicality through a real deployment, we implemented a prototype SEIZE switch using two open-source routing software platforms: *Click* [24] and *XORP* [25]. Figure 6 shows the overall structure of our implementation. SEIZE’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. Finally, we also extended Click to implement SEIZE’s data-plane logic, including consistent hashing and packet encapsulation. Our control and data plane modifications to Click are implemented as the *SeizeSwitch* element shown in Figure 6.

6.1 Maintaining switch-topology with XORP

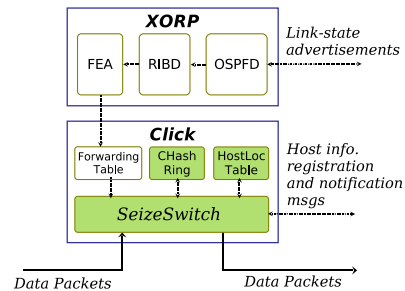


Figure 6: Implementation architecture.

XORP is an extensible open-source software router [25], which consists of several routing protocols and management functions. As previously mentioned, SEIZE relies on a link-state protocol to provide reachability between switches. We configured XORP’s OSPF protocol daemon to provide this function. In particular, we run a XORP OSPF process which contains a complete switch-level network map that is populated by exchanging LSAs with other switches. Based on this network map, the XORP RIBD (Routing Information Base Daemon) runs Dijkstra’s shortest-path algorithm to construct its own routing table. Once this routing table is created, RIBD installs the table into the forwarding plane process, which we implement with user-level Click. Click uses this finalized version of routing table (i.e., a forwarding table) to determine a next hop. The FEA (Forwarding Engine Abstraction) in XORP handles inter-process communication between XORP and Click.

However, XORP’s OSPF implementation assumes each

network node (router or switch) is identified by an IP address, as opposed to a MAC address. Because of this, our prototype does not automatically select its identifier from the MAC addresses assigned its interfaces. Instead, each switch is assigned a unique IP address on its loop-back interface, and advertises that address via a router LSA. Hence, when switches forward data packets across multiple hops (i.e., from ingress to egress or relay), they use Ethernet-in-IP encapsulation. We later plan to extend our implementation to perform Ethernet-in-Ethernet encapsulation for improved efficiency and simplicity. We note that other link-state protocols such as IS-IS [26] work with non-IP interfaces, but XORP does not currently support these protocols.

When our prototype switch is first started up, a script is run to detect the status of each network interface card. It then executes a simple neighbor-discovery protocol to determine which interfaces are connected to other switches, and over each of these interfaces it initiates an OSPF session and exchange LSAs. The link weight associated with the OSPF adjacency is by default set to be the link latency. If desired, another metric may be used. Meanwhile, the switch begins to receive packets through switch-to-host interfaces and learns end-hosts' information. All the processes constituting a SEIZE switch are spawned and managed by a single master process.

6.2 Building SEIZE control plane in Click

As previously mentioned, SEIZE employs several control messages to handle host information, including host registration, location notification for CTF, and host re-registration due to topology changes. We implemented this functionality as part of the Click process because most SEIZE control messages are triggered by data packets except the host re-registration due to topology changes. Specifically, we made two changes to Click.

First, in order to forward packets, the SEIZE switch must know the location of remote hosts. To deal with this, we implemented the HostLocTable module. The HostLocTable is populated with three kinds of host information: (a) the outbound port for every local host; (b) the locator (access switch's id) for every remote host for which this switch is a relay; and (c) the locator for every remote host cached via CTF. An insertion or deletion on this table takes place when a new host arrives, an existing host leaves, a host is registered with or deregistered, or cached host information is evicted from the switch. For each insertion or deletion of a locally-attached host, the SeizeSwitch generates a corresponding registration or deregistration message.

Second, we modified Click to monitor changes on the link-state map maintained by XORP, and to send registration messages when a host must change to a different relay. The Click process can handle this by monitoring the forwarding table that XORP installs/modifies when topology change occurs. Each entry in this forwarding table is a map between a switch's identifier s and the next hop switch n that should be used to reach s . Whenever the table changes, Click populates a consistent hash ring with every switch identifier

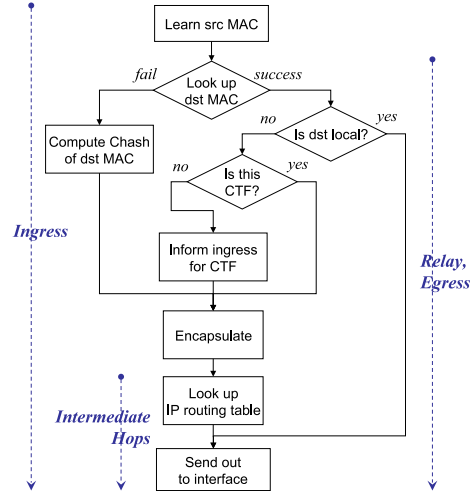


Figure 7: Packet processing flowchart.

found in the table. This hash ring allows for the Click data plane to perform consistent hashing to map a host to its relay. When topology change occurs and it alters the number of unique switch identifiers in this forwarding table, the SeizeSwitch first modifies the hash ring by adding/deleting the switch on the ring. If this change to the ring triggers hosts re-registration, the SeizeSwitch issues corresponding registration messages. For host registration and notification, we implemented a simple acknowledgment-based protocol to ensure reliable delivery.

6.3 Building SEIZE data plane in Click

The way the SEIZE data plane handles each packet is illustrated in Figure 7. Packet processing utilizes the three data structures we mentioned above: the forwarding table to determine a next hop, the consistent hash ring to map a host to a relay, and the HostLocTable to determine host location information.

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

Then the switch looks up the destination MAC address in the HostLocTable. If the look-up fails, the switch executes the consistent hash function \mathcal{F} with the destination MAC address. It then obtains a corresponding relay switch's identifier, encapsulates the packet with that identifier, and determines the next hop from the forwarding table. IP then delivers the packet via regular shortest-path forwarding. On the other hand, if the look-up succeeds, there are three possibilities: The switch knows the destination host because (i) the host is directly connected to it, (ii) the host is registered with the switch for relaying, or (iii) the host's location is cached at the switch due to CTF. In case (i), the switch simply sends out the packet to a corresponding out-

put port. In cases (ii) or (iii), the switch encapsulates the packet with the identifier of the destination’s access switch contained in the retrieved HostLocTable entry, and determine the next hop via the forwarding table. Additionally for (ii), the switch must send a host location notification message to the ingress for CTF. Note that this host location resolution procedure takes place only at ingress, relay, and egress switches. All the other intermediate switches simply forward packets based on the next-hop information from their forwarding table. SEIZE 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 would provide the same functionality.

7. Implementation Results

In this section we report performance results from a deployment of our prototype implementation on Emulab [27]. First, we compare performance with our simulator to validate correctness. Next, we present a set of microbenchmarks to evaluate per-packet processing overheads. Then, to evaluate dynamics of a SEIZE network, we measure control overhead and switch state requirements over time. Finally, we investigate impact of SEIZE on application-level performance, through a combination of web benchmarks [28] and TCP throughput measurements.

Cross-validation: To ensure simulation results collected in the previous section would roughly characterize performance of a real SEIZE 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 SEIZE. 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 1 shows the per-packet processing time for both SEIZE and Ethernet. We measure this as the time from when a packet enters the switch’s inbound queue, to the time it departs. We break this time down into the major components shown in Figure 7. From the table, we can see that ingress and relay switches in SEIZE require more processing time than Ethernet. This happens because SEIZE has to encapsulate a packet and then look up the forwarding table with the outer header. When a destination location is unknown at ingresses, our implementation also requires to run a particularly heavyweight MD5 hash. We chose MD5 only because it was widely used. Since SEIZE does not rely on MD5’s security properties, we plan to replace it with one of the much more efficient software hashing algorithms or hardware implementations as part of future work. However, SEIZE requires less packet processing overhead than Ethernet at other hops on a path except

an ingress (and a relay when CTF is disabled). Because of this, we found that SEIZE with CTF requires less overall processing time on paths longer than 3.03 switch-level hops. As a comparison point, we found the average number of switch-level hops between hosts in a real university campus network (*Entp-campus*) to be over 4 for the vast majority of host pairs.

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

	<i>learn src</i>	<i>look-up host tbl</i>	<i>chash</i>	<i>encap</i>	<i>look-up fwd tbl</i>	<i>Total</i>
<i>SEIZE-ingress (dst loc unknown)</i>	0.61	0.63	2.22	0.67	0.62	4.75
<i>SEIZE-ingress (dst loc known)</i>	0.61	0.63	-	0.67	0.62	2.53
<i>SEIZE-relay</i>	-	0.63	-	0.67	0.62	1.92
<i>SEIZE-egress</i>	-	0.63	-	-	-	0.63
<i>SEIZE-others</i>	-	-	-	-	0.67	0.67
<i>Ethernet</i>	0.63	0.64	-	-	-	1.27

Effect of network dynamics: To evaluate the dynamics of SEIZE and Ethernet, we instrumented the switch’s internal data structures to periodically measure performance information. Figures 8a and 8b show control overhead and forwarding-table size, respectively, measured over one-second intervals. We can see that SEIZE has much lower control overhead when the systems are first started up. However, SEIZE’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 SEIZE 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. Finally, we investigate performance by injecting host scanning attacks [5]. Figure 8c shows control overhead where a single host scans 5000 random destination hosts that do not exist in the network, at 300 and 600 seconds into the trace. In Ethernet, every scanning packet sent to a destination generates a network-wide flood unless the destination’s MAC address is already cached. This results in increased control overhead when hosts scan. In SEIZE, each scan generates one unicast packet to the relay for every scanned host. The SEIZE relay then forwards the packet to the destination. If the scan targets destinations that do not exist in the network, the packet would be discarded at the relay.

Impact on application-level performance: We evaluated SEIZE’s effect on application traffic by performing web performance benchmarks. Here, we configure a topology consisting of four switches connected in a full mesh via links with 10 to 20 msec of delays. We place four hosts *h0*, *h1*, *h2*, *h3*, one at each of the four switches. We then use the *Flexiclient* web benchmark tool [28] to generate a collection of test files and request patterns, and measure the amount of time required for hosts to download these files using both SEIZE and Ethernet. Table 2 shows this time with three file sizes ranging from 5KB to 500KB over 50 trials. At first, one might think that SEIZE with CTF may increase download latency, as the first few packets may traverse a differ-

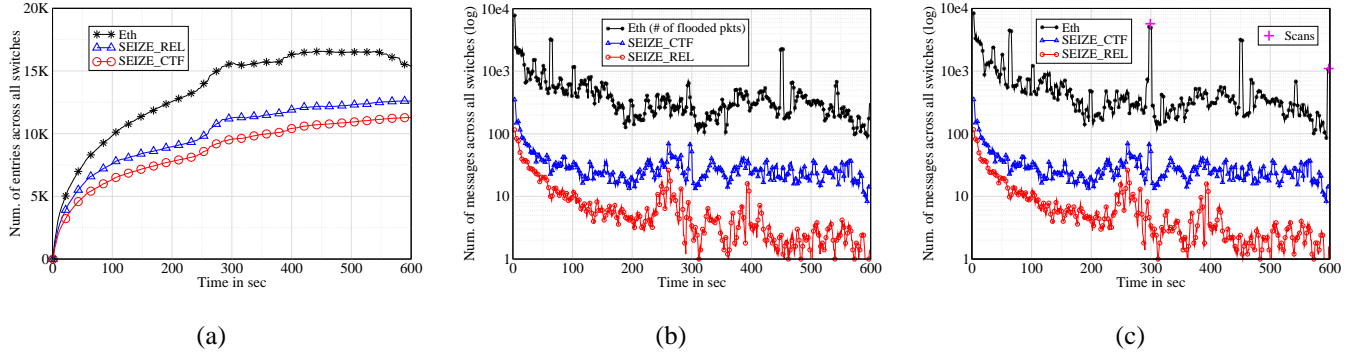


Figure 8: Effect of network dynamics: (a) table size (b) control overhead (c) control overhead during a scanning attack.

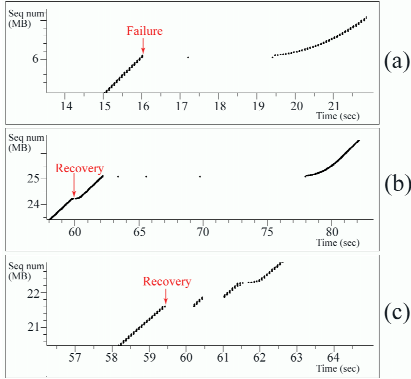


Figure 9: Fail-over performance: Impact on a TCP flow by a: (a) switch failure, (b) switch recovery (before optimization), (c) switch recovery (after optimization).

ent path from later packets, possibly resulting in reordering. However, as can be seen from the table, this effect is negligible, and has little effect of download time. Note that the performance of Ethernet in the case of $h1$ and $h3$ is worse than that of SEIZE because traffic between $h0$ and $h1/h3$ is forwarded through the root bridge $h2$.

Table 2: Web performance benchmarks, download time per file [sec].

	SEIZE-CTF			SEIZE-REL			Ethernet		
	h1	h2	h3	h1	h2	h3	h1	h2	h3
5KB	0.07	0.10	0.06	0.07	0.11	0.06	0.15	0.09	0.10
50KB	0.15	0.22	0.11	0.15	0.26	0.11	0.35	0.21	0.23
500KB	0.57	0.82	0.42	0.57	0.99	0.42	1.36	0.81	0.88

Fail-over performance: Figure 9 shows the effect of switch failure on application level performance. Here we disable CTF, induce failures of the relay switch, and measure throughput of TCP when all packets are forwarded through the relay. We set the OSPF hello interval to 1 second, and dead interval to 3 seconds. After the relay fails, there is some convergence delay before packets are sent via the new relay. We found that SEIZE restores connectivity quickly, typically on the order of several hundred milliseconds after the dead interval. This allows TCP to recover within several seconds in total, as shown in Figure 9a. 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 switch. Figure 9b shows TCP can re-

cover in several seconds from switch repairs as well. We found performance during recoveries could be improved by continuing to forward packets through the old relay for a grace period, as shown in Figure 9c. In general, optimizing an Ethernet network to attain low convergence delay exposes the network to a high chance of broadcast storms, making it nearly impossible to realize in a large network.

8. Related Work

There are two main bodies of related work. First, there has been work on improving efficiency of network forwarding. SmartBridges [29] and RBridges [14] leverage a link-state protocol to disseminate information about bridge connectivity. This eliminates the need to maintain a spanning tree and improves forwarding paths. CMU-Ethernet [8] also leverages link-state, but replaces per-host broadcasting by placing host information in LSAs. Viking [6] used multiple spanning trees for faster fault recovery, which can be dynamically adjusted to conform to changing load. However, all these four approaches rely on network-wide floods to disseminate information about hosts, which may interfere with scaling to large networks.

Second, there has been work on using hashing to support flat addressing. Ray et al.’s resolution scheme [30] was done in parallel with ours [1] and also eliminates network-wide broadcast with a hash-based mechanism. However, it continues to rely on a spanning tree for forwarding, which may lead to imbalanced load and suboptimal paths. In addition, VRR [31], ROFL [32] and UIP [33] rely on building a network-layer Distributed Hash Table (DHT), and using the DHT structure to route. Of these, VRR primarily targeted wireless ad-hoc networks, and UIP targeted NAT bridging, and ROFL extended these techniques to operate at the network layer. ROFL faces scaling and performance issues in large networks, primarily in terms of router state requirements and stretch [32]. Moreover, SEIZE does not require nodes to consistently manage a multi-hop DHT structure, which may simplify some aspects of troubleshooting and engineering traffic. Instead, SEIZE leverages the network-wide view given by a link-state routing protocol to provide a one-hop DHT [3]. This ensures switches can process resolution queries via a single look-up operation on the hash space,

which in turn allows to provide much stronger bounds on latency as compared to protocols like ROFL.

9. 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 SEIZE takes an important first step towards solving these problems, by providing scalable self-configuring routing. SEIZE 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, SEIZE also provides add-ons for administrators who wish to customize network operation.

Experiments with our initial prototype implementation show that SEIZE 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 currently extending our implementation into kernel-level Click, with the goal to improve packet processing speeds and reaction time to network changes. In addition, we are interested in investigating the deployability of SEIZE. We are also interested in ramifications of SEIZE on switch architectures, and how to design switch hardware to efficiently support SEIZE. 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.

10. References

- [1] C. Kim and J. Rexford, "Revisiting Ethernet: Plug-and-play made scalable and efficient," in *Proc. IEEE LANMAN*, June 2007. invited paper (short workshop paper).
- [2] E. Brewer, M. Demmer, M. Ho, R. Honicky, J. Pal, M. Plauche, and S. Surana, "The challenges of technology research for developing regions," in *Proc. Pervasive Computing*, April-June 2006.
- [3] A. Gupta, B. Liskov, and R. Rodrigues, "Efficient routing for peer-to-peer overlays," in *Proc. NSDI*, March 2004.
- [4] W. Aiello, C. Kalmanek, P. McDaniel, S. Sen, O. Spatscheck, and J. van der Merwe, "Analysis of communities of interest in data networks," in *Proc. Passive and Active Measurement*, March 2005.
- [5] M. Allman, V. Paxson, and J. Terrell, "A brief history of scanning," in *Proc. Internet Measurement Conference*, October 2007.
- [6] S. Sharma, K. Gopalan, S. Nanda, and T. Chiueh, "Viking: A multi-spanning-tree Ethernet architecture for metropolitan area and cluster networks," in *Proc. IEEE INFOCOM*, March 2004.
- [7] "IEEE Std 802.1Q - 2005, IEEE Standard for Local and Metropolitan Area Network, Virtual Bridged Local Area Networks," 2005. standards.ieee.org/getieee802/download/802.1Q-2005.pdf.
- [8] A. Myers, E. Ng, and H. Zhang, "Rethinking the service model: scaling Ethernet to a million nodes," in *Proc. HotNets*, November 2004.
- [9] Dartmouth Institute for Security Technology Studies, "Problems with broadcasts," http://www.ists.dartmouth.edu/classroom/crs/arp_broadcast.php.
- [10] B. Albrightson, J. Garcia-Luna-Aceves, and J. Boyle, "EIGRP – a fast routing protocol based on distance vectors," in *Proc. Network/Interop*, May 1994.
- [11] J. Moy, *OSPF: Anatomy of an Internet Routing Protocol*. Addison-Wesley, 1998.
- [12] Z. Kerravala, "Configuration management delivers business resiliency," November 2002. The Yankee Group.
- [13] R. King, "Traffic management tools fight growing pains," June 2004. www.thewhir.com/features/traffic-management.cfm.
- [14] R. Perlman, "Rbridges: Transparent routing," in *Proc. IEEE INFOCOM*, March 2004.
- [15] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load balancing in dynamic structured p2p systems," in *Proc. IEEE INFOCOM*, March 2003.
- [16] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. ACM Symposium on Theory of Computing*, 1997.
- [17] P. White, "ATM switching and IP routing integration: the next stage in Internet evolution?," in *IEEE Communication Magazine*, April 1998.
- [18] R. Droms, "Dynamic Host Configuration Protocol." Request for Comments 2131, March 1997.
- [19] R. Perlman, *Interconnections: Bridges, routers, switches, and internetworking protocols*. Addison-Wesley, second ed., 1999.
- [20] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney, "A first look at modern enterprise traffic," in *Proc. Internet Measurement Conference*, October 2005.
- [21] N. Spring, R. Mahajan, and D. Wetherall, "Measuring ISP topologies with Rocketfuel," in *Proc. ACM SIGCOMM*, August 2002.
- [22] "Gigabit campus network design – principles and architecture," 1999. www.cisco.com/warp/public/cc/so/neso/l/ncso/gcnd_wp.pdf.
- [23] "Skitter." <http://www.caida.org/tools/measurement/skitter>.
- [24] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek, "The Click modular router," in *ACM Trans. Computer Systems*, August 2000.
- [25] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov, "Designing extensible IP router software," in *Proc. NSDI*, May 2005.
- [26] D. Oran, "OSI IS-IS Intra-domain routing protocol." RFC 1142, February 1990.
- [27] B. White, J. Lepreau, L. Stoller, R. Ricci, G. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *Proc. OSDI*, December 2002.
- [28] V. Pai, P. Druschel, and W. Zwaenepoel, "Flash: An efficient and portable web server," in *Proc. USENIX Annual Technical Conference*, June 1999.
- [29] T. Rodeheffer, C. Thekkath, and D. Anderson, "SmartBridge: A scalable bridge architecture," in *Proc. ACM SIGCOMM*, August 2000.
- [30] S. Ray, R. A. Guerin, and R. Sofi a, "A distributed hash table based address resolution scheme for large-scale Ethernet networks," in *Proc. International Conference on Communications*, 2007. June.
- [31] M. Caesar, M. Castro, E. Nightingale, A. Rowstron, and G. O'Shea, "Virtual Ring Routing: Network routing inspired by DHTs," in *Proc. ACM SIGCOMM*, September 2006.
- [32] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, and I. Stoica, "ROFL: Routing on Flat Labels," in *Proc. ACM SIGCOMM*, September 2006.
- [33] B. Ford, "Unmanaged Internet Protocol: Taming the edge network management crisis," in *Proc. HotNets*, November 2003.