Ethane: Taking Control of the Enterprise

Martìn Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, and Nick McKeown
Stanford University

Scott Shenker
U.C. Berkeley and ICSI

ABSTRACT

This paper presents Ethane, a new network architecture for the enterprise. Ethane allows managers to define a single network-wide fine-grain policy, and then enforces it directly. Ethane couples extremely simple flow-based Ethernet switches with a centralized controller that manages the admittance and routing of flows. While radical, this design is backwards-compatible with existing hosts and switches.

We have implemented Ethane in both hardware and software, supporting both wired and wireless hosts. Our operational Ethane network has supported over 300 hosts for the past four months in in Stanford University's network, and this deployment experience has significantly affected Ethane's design.

Categories and Subject Descriptors

C.2.6 [**Computer Communication Networks**]: Internetworking; C.2.1 [**Computer Communication Networks**]: Network Architecture and Design

General Terms

Design, Experimentation, Performance

Keywords

Network, Architecture, Security, Management

1. INTRODUCTION

Enterprise networks are often large, run a wide variety of applications and protocols, and typically operate under strict reliability and security constraints; thus, they represent a challenging environment for network management. The stakes are high, as business productivity can be severely hampered by network misconfigurations or break-ins. Yet the current solutions are weak, making enterprise network management both expensive and error-prone. Indeed, most networks today require substantial manual configuration by trained operators [11, 22, 23, 25] to achieve even moderate security [24]. A Yankee Group report found that 62% of network downtime in multi-vendor networks comes from human-error and that 80% of IT budgets is spent on maintenance and operations [16].

There have been many attempts to make networks more manageable and more secure. One approach introduces proprietary middleboxes that can exert their control effectively only if placed at network choke-points. If traffic accidentally flows (or is maliciously diverted) around the middlebox, the network is no longer managed nor secure [25]. Another approach is to add functionality to existing networks—to provide tools for diagnosis, to offer controls for VLANs, access-control lists, and filters to isolate users, to instrument the routing and spanning tree algorithms to support better connectivity management, and then to collect packet traces to allow auditing. This can be done by adding a new layer of protocols, scripts, and applications [1, 10] that help automate configuration management in order to reduce the risk of errors. However, these solutions hide the complexity, not reduce it. And they have to be constantly maintained to support the rapidly changing and often proprietary management interfaces exported by the managed elements.

Rather than building a new layer of complexity on top of the network, we explore the question: *How could we change the enterprise network architecture to make it more manageable?* Our answer is embodied in the architecture we describe here, called Ethane. Ethane is built around three fundamental principles that we feel are important to any network management solution:

The network should be governed by policies declared over high-level names. Networks are most easily managed in terms of the entities we seek to control—such as users, hosts, and access points—rather than in terms of low-level and often dynamically-allocated addresses. For example, it is convenient to declare which services a user is allowed to use and to which machines they can connect.

Policy should determine the path that packets follow. There are several reasons for policy to dictate the paths. First, policy might require packets to pass through an intermediate middlebox; for example, a guest user might be required to communicate via a proxy, or the user of an unpatched operating system might be required to communicate via an intrusion detection system. Second, traffic can receive more appropriate service if its path is controlled; directing real-time communications over lightly loaded paths, important communications over redundant paths, and private communications over paths inside a trusted boundary would all lead to better service. Allowing the network manager to determine the paths via policy—where the policy is in terms of high-level names—leads to finer-level control and greater visibility than is easily achievable with current designs.

The network should enforce a strong binding between a packet and its origin. Today, it is notoriously difficult to reliably determine the origin of a packet: Addresses are dynamic and change

SIGCOMM'07, August 27–31, 2007, Kyoto, Japan.

frequently, and they are easily spoofed. The loose binding between users and their traffic is a constant target for attacks in enterprise networks. If the network is to be governed by a policy declared over high-level names (*e.g.*, users and hosts) then packets should be identifiable, without doubt, as coming from a particular physical entity. This requires a strong binding between a user, the machine they are using, and the addresses in the packets they generate. This binding must be kept consistent at all times, by tracking users and machines as they move.

To achieve these aims, we followed the lead of the 4D project [14] and adopted a centralized control architecture. Centralized solutions are normally an anathema for networking researchers, but we feel it is the proper approach for enterprise management. IP's best-effort service model is both simple and unchanging, well-suited for distributed algorithms. Network management is quite the opposite; its requirements are complex and require strong consistency, making it quite hard to compute in a distributed manner.

There are many standard objections to centralized approaches, such as resilience and scalability. However, as we discuss later in the paper, our results suggest that standard replication techniques can provide excellent resilience, and current CPU speeds make it possible to manage all control functions on a sizable network (*e.g.*, 25,000 hosts) from a single commodity PC.

Ethane bears substantial resemblance to SANE, our recently-proposed clean-slate approach to enterprise security [12]. SANE was, as are many clean-slate designs, difficult to deploy and largely untested. While SANE contained many valuable insights, Ethane extends this previous work in three main ways:

Security follows management. Enterprise security is, in many ways, a subset of network management. Both require a network policy, the ability to control connectivity, and the means to observe network traffic. Network management wants these features so as to control and isolate resources, and then to diagnose and fix errors, whereas network security seeks to control who is allowed to talk to whom, and then to catch bad behavior before it propagates. When designing Ethane, we decided that a broad approach to network management would also work well for network security.

Incremental deployability. SANE required a "fork-lift" replacement of an enterprise's entire networking infrastructure and changes to all the end-hosts. While this might be suitable in some cases, it is clearly a significant impediment to widespread adoption. Ethane is designed so that it can be incrementally deployed within an enterprise: it does not require any host modifications, and Ethane Switches can be incrementally deployed alongside existing Ethernet switches.

Significant deployment experience. Ethane has been implemented in both software and hardware (special-purpose Gigabit Ethernet switches) and deployed at Stanford's Computer Science department for over four months and managed over 300 hosts. This deployment experience has given us insight into the operational issues such a design must confront, and resulted in significant changes and extensions to the original design.

In this paper, we describe our experiences designing, implementing, and deploying Ethane. We begin with a high-level overview of the Ethane design in §2, followed by a detailed description in §3. In §4, we describe a policy language *Pol-Eth* that we built to manage our Ethane implementation. We then discuss our implementation and deployment experience (§5), followed by performance analysis (§6). Finally we present limitations (§7), discuss related work (§8), and then conclude (§9).

2. OVERVIEW OF ETHANE DESIGN

Ethane controls the network by not allowing any communication between end-hosts without explicit permission. It imposes this requirement through two main components. The first is a central *Controller* containing the global network policy that determines the fate of all packets. When a packet arrives at the Controller—how it does so is described below—the Controller decides whether the flow represented by that packet[1] should be allowed. The Controller knows the global network topology and performs route computation for permitted flows. It grants access by explicitly enabling flows within the network switches along the chosen route. The Controller can be replicated for redundancy and performance.

The second component is a set of Ethane *Switches*. In contrast to the omniscient Controller, these Switches are simple and dumb. Consisting of a simple flow table and a secure channel to the Controller, Switches simply forward packets under the direction of the Controller. When a packet arrives that is not in the flow table, they forward that packet to the Controller (in a manner we describe later), along with information about which port the packet arrived on. When a packet arrives that is in the flow table, it is forwarded according to the Controller's directive. Not every switch in an Ethane network needs to be an Ethane Switch: Our design allows Switches to be added gradually, and the network becomes more manageable with each additional Switch.

2.1 Names, Bindings, and Policy Language

When the Controller checks a packet against the global policy, it is evaluating the packet against a set of simple rules, such as "Guests can communicate using HTTP, but only via a web proxy" or "VoIP phones are not allowed to communicate with laptops." If we want the global policy to be specified in terms of such physical entities, we need to reliably and securely associate a packet with the user, group, or machine that sent it. If the mappings between machine names and IP addresses (DNS) or between IP addresses and MAC addresses (ARP and DHCP) are handled elsewhere and are unauthenticated, then we cannot possibly tell who sent the packet, even if the user authenticates with the network. This is a notorious and widespread weakness in current networks.

With (logical) centralization, it is simple to keep the namespace consistent as components join, leave and move around the network. Network state changes simply require updating the bindings at the Controller. This is in contrast to today's network where there are no widely used protocols for keeping this information consistent. Further, distributing the namespace among all switches would greatly increase the trusted computing base and require high overheads to maintain consistency on each bind event.

In Ethane, we also use a sequence of techniques to secure the bindings between packet headers and the physical entities that sent them. First, Ethane takes over all the binding of addresses. When machines use DHCP to request an IP address, Ethane assigns it knowing to which switch port the machine is connected, enabling Ethane to attribute an arriving packet to a physical port.[2] Second, the packet must come from a machine that is registered on the network, thus attributing it to a particular machine. Finally, users are required to authenticate themselves with the network—for exam-

[1] All policies considered in Ethane are based over flows, where the header fields used to define a flow are based on the packet type (for example, TCP/UDP flows include the Ethernet, IP and transport headers). Thus, only a single policy decision need be made for each such "flow".

[2] As we discuss later, a primary advantage of knowing the ingress port of a packet is that it allows the Controller to apply filters to the first-hop switch used by unwanted traffic.

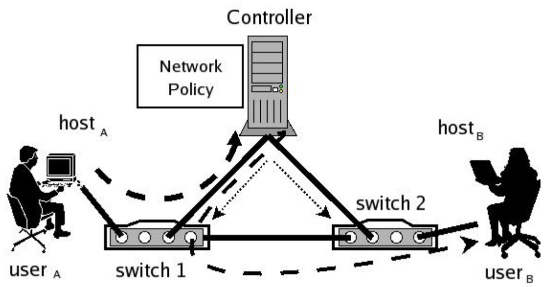

Figure 1: Example of communication on an Ethane network. Route setup shown by dotted lines; the path taken by the first packet of a flow shown by dashed lines.

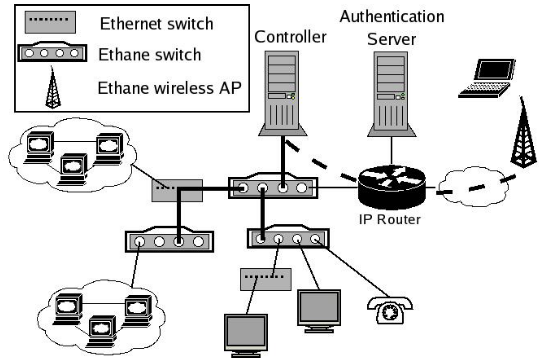

Figure 2: An example Ethane deployment.

ple, via HTTP redirects in a manner similar to those used by commercial WiFi hotspots—binding users to hosts. Therefore, whenever a packet arrives at the Controller, it can securely associate the packet to the particular user and host that sent it.

There are several powerful consequences of the Controller knowing both where users and machines are attached and all bindings associated with them. First, the Controller can keep track of where any entity is located: When it moves, the Controller finds out as soon as packets start to arrive from a different Switch port. The Controller can choose to allow the new flow or it might choose to deny the moved flow (*e.g.*, to restrict mobility for a VoIP phone due to E911 regulations). Another powerful consequence is that the Controller can journal all bindings and flow-entries in a log. Later, if needed, the Controller can reconstruct all network events; *e.g.*, which machines tried to communicate or which user communicated with a service. This can make it possible to diagnose a network fault or to perform auditing or forensics, long after the bindings have changed.

In principle, Ethane does not mandate the use of a particular policy language. For completeness, however, we have designed and deployed *Pol-Eth*, in which policies are declared as a set of rules consisting of predicates and, for matching flows, the set of resulting actions (*e.g.*, allow, deny, or route via a waypoint). As we will see, *Pol-Eth*'s small set of easily understood rules can still express powerful and flexible policies for large, complex networks.

2.2 Ethane in Use

Putting all these pieces together, we now consider the five basic activities that define how an Ethane network works, using Figure 1 to illustrate:

Registration. All Switches, users, and hosts are registered at the Controller with the credentials necessary to authenticate them. The credentials depend on the authentication mechanisms in use. For example, hosts may be authenticated by their MAC addresses, users via username and password, and switches through secure certificates. All switches are also preconfigured with the credentials needed to authenticate the Controller (*e.g.*, the Controller's public key).

Bootstrapping. Switches bootstrap connectivity by creating a spanning tree rooted at the Controller. As the spanning tree is being created, each switch authenticates with and creates a secure channel to the Controller. Once a secure connection is established, the switches send link-state information to the Controller, which aggregates this information to reconstruct the network topology.

Authentication.

1. User$_A$ joins the network with host$_A$. Because no flow entries exist in switch 1 for the new host, it will initially forward all of host$_A$'s packets to the Controller (marked with switch 1's ingress port).
2. Host$_A$ sends a DHCP request to the Controller. After checking host$_A$'s MAC address,[3] the Controller allocates an IP address (IP$_A$) for it, binding host$_A$ to IP$_A$, IP$_A$ to MAC$_A$, and MAC$_A$ to a physical port on switch 1.
3. User$_A$ opens a web browser, whose traffic is directed to the Controller, and authenticates through a web-form. Once authenticated, user$_A$ is bound to host$_A$.

Flow Setup.

1. User$_A$ initiates a connection to user$_B$ (who we assume has already authenticated in a manner similar to user$_A$). Switch 1 forwards the packet to the Controller after determining that the packet does not match any active entries in its flow table.
2. On receipt of the packet, the Controller decides whether to allow or deny the flow, or require it to traverse a set of waypoints.
3. If the flow is allowed, the Controller computes the flow's route, including any policy-specified waypoints on the path. The Controller adds a new entry to the flow tables of all the Switches along the path.

Forwarding.

1. If the Controller allowed the path, it sends the packet back to switch 1 which forwards it based on the new flow entry. Subsequent packets from the flow are forwarded directly by the Switch, and are not sent to the Controller.
2. The flow-entry is kept in the switch until it times out (due to inactivity) or is revoked by the Controller.

3. ETHANE IN MORE DETAIL

3.1 An Ethane Network

Figure 2 shows a typical Ethane network. The end-hosts are unmodified and connect via a wired Ethane Switch or an Ethane wireless access point. (From now on, we will refer to both as "Switches", described next in §3.2).[4]

[3] The network may use a stronger form of host authentication, such as 802.1X, if desired.

[4] We will see later that an Ethane network can also include legacy Ethernet switches and access points, so long as we include some Ethane Switches in the network. The more switches we replace, the easier to manage and the more secure the network.

When we add an Ethane Switch to the network, it has to find the Controller (§3.3), open a secure channel to it, and help the Controller figure out the topology. We do this with a modified minimum spanning tree algorithm (per §3.7 and denoted by thick, solid lines in the figure). The outcome is that the Controller knows the whole topology, while each Switch only knows a part of it.

When we add (or boot) a host, it has to authenticate itself with the Controller. From the Switch's point-of-view, packets from the new host are simply part of a new flow, and so packets are automatically forwarded to the Controller over the secure channel, along with the ID of the Switch port on which they arrived. The Controller authenticates the host and allocates its IP address (the Controller includes a DHCP server).

3.2 Switches

A wired Ethane Switch is like a simplified Ethernet switch. It has several Ethernet interfaces that send and receive standard Ethernet packets. Internally, however, the switch is much simpler, as there are several things that conventional Ethernet switches do that an Ethane switch doesn't need: An Ethane Switch doesn't need to learn addresses, support VLANs, check for source-address spoofing, or keep flow-level statistics (*e.g.*, start and end time of flows, although it will typically maintain per-flow packet and byte counters for each flow entry). If the Ethane Switch is replacing a Layer-3 "switch" or router, it doesn't need to maintain forwarding tables, ACLs, or NAT. It doesn't need to run routing protocols such as OSPF, ISIS, and RIP. Nor does it need separate support for SPANs and port-replication (this is handled directly by the flow table under the direction of the Controller).

It is also worth noting that the flow table can be several orders-of-magnitude smaller than the forwarding table in an equivalent Ethernet switch. In an Ethernet switch, the table is sized to minimize broadcast traffic: as switches flood during learning, this can swamp links and makes the network less secure.[5] As a result, an Ethernet switch needs to remember all the addresses it's likely to encounter; even small wiring closet switches typically contain a million entries. Ethane Switches, on the other hand, can have much smaller flow tables: they only need to keep track of flows in-progress. For a wiring closet, this is likely to be a few hundred entries at a time, small enough to be held in a tiny fraction of a switching chip. Even for a campus-level switch, where perhaps tens of thousands of flows could be ongoing, it can still use on-chip memory that saves cost and power.

We expect an Ethane Switch to be far simpler than its corresponding Ethernet switch, without any loss of functionality. In fact, we expect that a large box of power-hungry and expensive equipment will be replaced by a handful of chips on a board.

Flow Table and Flow Entries. The Switch datapath is a managed flow table. Flow entries contain a Header (to match packets against), an Action (to tell the switch what to do with the packet), and Per-Flow Data (which we describe below).

There are two common types of entry in the flow table: per-flow entries describing application flows that should be *forwarded*, and per-host entries that describe misbehaving hosts whose packets should be *dropped*. For TCP/UDP flows, the Header field covers the TCP/UDP, IP, and Ethernet headers, as well as physical port information. The associated Action is to forward the packet to a particular interface, update a packet-and-byte counter (in the Per-Flow Data), and set an activity bit (so that inactive entries can be timed-out). For misbehaving hosts, the Header field contains an Ethernet source address and the physical ingress port.[6] The associated Action is to drop the packet, update a packet-and-byte counter, and set an activity bit (to tell when the host has stopped sending).

Only the Controller can add entries to the flow table. Entries are removed because they timeout due to inactivity (local decision) or because they are revoked by the Controller. The Controller might revoke a single, badly behaved flow, or it might remove a whole group of flows belonging to a misbehaving host, a host that has just left the network, or a host whose privileges have just changed.

The flow table is implemented using two exact-match tables: One for application-flow entries and one for misbehaving-host entries. Because flow entries are exact matches, rather than longest-prefix matches, it is easy to use hashing schemes in conventional memories rather than expensive, power-hungry TCAMs.

Other Actions are possible in addition to just *forward* and *drop*. For example, a Switch might maintain multiple queues for different classes of traffic, and the Controller can tell it to queue packets from application flows in a particular queue by inserting queue IDs into the flow table. This can be used for end-to-end L2 isolation for classes of users or hosts. A Switch could also perform address translation by replacing packet headers. This could be used to obfuscate addresses in the network by "swapping" addresses at each Switch along the path—an eavesdropper would not be able to tell which end-hosts are communicating—or to implement address translation for NAT in order to conserve addresses. Finally, a Switch could control the rate of a flow.

The Switch also maintains a handful of implementation-specific entries to reduce the amount of traffic sent to the Controller. This number should remain small to keep the Switch simple, although this is at the discretion of the designer. On one hand, such entries can reduce the amount of traffic sent to the Controller; on the other hand, any traffic that misses on the flow table will be sent to the Controller anyway, so this is just an optimization.

Local Switch Manager. The Switch needs a small local manager to establish and maintain the secure channel to the Controller, to monitor link status, and to provide an interface for any additional Switch-specific management and diagnostics. (We implemented our manager in the Switch's software layer.)

There are two ways a Switch can talk to the Controller. The first one, which we have assumed so far, is for Switches that are part of the same physical network as the Controller. We expect this to be the most common case; *e.g.*, in an enterprise network on a single campus. In this case, the Switch finds the Controller using our modified Minimum Spanning Tree protocol described in §3.7. The process results in a secure channel stretching through these intermediate Switches all the way to the Controller.

If the Switch is not within the same broadcast domain as the Controller, the Switch can create an IP tunnel to it (after being manually configured with its IP address). This approach can be used to control Switches in arbitrary locations, *e.g.*, the other side of a conventional router or in a remote location. In one application of Ethane, the Switch (most likely a wireless access point) is placed in a home or small business and then managed remotely by the Controller over this secure tunnel.

The local Switch manager relays link status to the Controller so it can reconstruct the topology for route computation. Switches maintain a list of neighboring switches by broadcasting and receiving neighbor-discovery messages. Neighbor lists are sent to the Controller after authentication, on any detectable change in link status, and periodically every 15 seconds.

[5] In fact, network administrators often use manually configured and inflexible VLANs to reduce flooding.

[6] If a host is spoofing, its first-hop port can be shut of directly (§3.3).

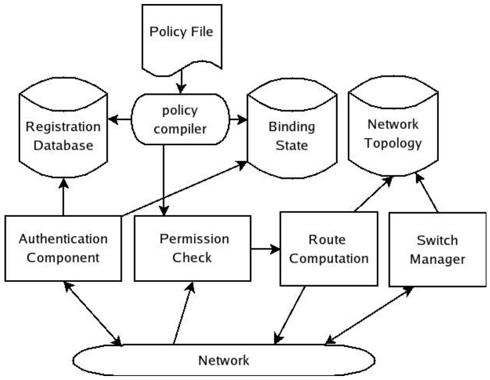

Figure 3: High-level view of Controller components.

3.3 Controller

The Controller is the brain of the network and has many tasks; Figure 3 gives a block-diagram. The components do not have to be co-located on the same machine (indeed, they are not in our implementation).

Briefly, the components work as follows. The *authentication component* is passed all traffic from unauthenticated or unbound MAC addresses. It authenticates users and hosts using credentials stored in the registration database. Once a host or user authenticates, the Controller remembers to which switch port they are connected.

The Controller holds the *policy file*, which is compiled into a fast lookup table (see §4). When a new flow starts, it is checked against the rules to see if it should be accepted, denied, or routed through a waypoint. Next, the *route computation* uses the network topology to pick the flow's route. The topology is maintained by the *switch manager*, which receives link updates from the Switches.

In the remainder of this section, we describe each component's function in more detail. We leave description of the policy language for the next section.

Registration. All entities that are to be named by the network (*i.e.*, hosts, protocols, Switches, users, and access points[7]) must be registered. The set of registered entities make up the policy namespace and is used to statically check the policy (§4) to ensure it is declared over valid principles.

The entities can be registered directly with the Controller, or—as is more likely in practice and done in our own implementation—Ethane can interface with a global registry such as LDAP or AD, which would then be queried by the Controller.

By forgoing Switch registration, it is also possible for Ethane to provide the same "plug-and-play" configuration model for Switches as Ethernet. Under this configuration, the Switches distribute keys on boot-up (rather than require manual distribution) under the assumption that the network has not been compromised.

Authentication. All Switches, hosts, and users must authenticate with the network. Ethane does not specify a particular host authentication mechanism; a network could support multiple authentication methods (*e.g.*, 802.1X or explicit user login) and employ entity-specific authentication methods. In our implementation, for example, hosts authenticate by presenting registered MAC addresses, while users authenticate through a web front-end to a Kerberos server. Switches authenticate using SSL with server- and client-side certificates.

Tracking Bindings. One of Ethane's most powerful features is that it can easily track all the bindings between names, addresses, and physical ports on the network, even as Switches, hosts, and users join, leave, and move around the network. It is Ethane's ability to track these dynamic bindings that makes the policy language possible: It allows us to describe policies in terms of users and hosts, yet implement the policy using flow tables in Switches.

A binding is never made without requiring authentication, so as to prevent an attacker from assuming the identity of another host or user. When the Controller detects that a user or host leaves, all of its bindings are invalidated, and all of its flows are revoked at the Switch to which it was connected. Unfortunately, in some cases, we cannot get reliable join and leave events from the network. Therefore, the Controller may resort to timeouts or the detection of movement to another physical access point before revoking access.

Namespace Interface. Because Ethane tracks all the bindings between users, hosts, and addresses, it can make information available to network managers, auditors, or anyone else who seeks to understand who sent what packet and when.

In current networks, while it is possible to collect packet traces, it is almost impossible to figure out later which user—or even which host—sent or received the packets, as the addresses are dynamic and there is no known relationship between users and packet addresses.

An Ethane Controller can journal all the authentication and binding information: The machine a user is logged in to, the Switch port their machine is connected to, the MAC address of their packets, and so on. Armed with a packet trace and such a journal, it is possible to determine exactly which user sent a packet, when it was sent, the path it took, and its destination. Obviously, this information is very valuable for both fault diagnosis and identifying break-ins. On the other hand, the information is sensitive and controls need to be placed on who can access it. We expect Ethane Controllers to provide an interface that gives privileged users access to the information. In our own system, we built a modified DNS server that accepts a query with a timestamp, and returns the complete bound namespace associated with a specified user, host, or IP address (described in §5).

Permission Check and Access Granting. Upon receiving a packet, the Controller checks the policy to see what actions apply to it. The results of this check (if the flow is allowed) are forwarded to the route computation component which determines the path given the policy constraint. In our implementation all paths are pre-computed and maintained via a dynamic all-pairs shortest path algorithm [13]. Section 4 describes our policy model and implementation.

Enforcing Resource Limits. There are many occasions when a Controller wants to limit the resources granted to a user, host, or flow. For example, it might wish to limit a flow's rate, limit the rate at which new flows are setup, or limit the number of IP addresses allocated. Such limits will depend on the design of the Controller and Switch, and they will be at the discretion of the network manager. In general, however, Ethane makes it easy to enforce such limits either by installing a filter in a Switch's flow table or by telling the Switch to limit a flow's rate.

The ability to directly manage resources from the Controller is the primary means of protecting the network (and Controller) from resource exhaustion attacks. To protect itself from connection flooding from unauthenticated hosts, a Controller can place a limit on the number of authentication requests per host and per switch port; hosts that exceed their allocation can be closed down by adding an entry in the flow table that blocks their MAC address. If such hosts

[7]We define an access point here as a {Switch,port} pair

spoof their address, the Controller can disable their Switch port. A similar approach can be used to prevent flooding from authenticated hosts.

Flow state exhaustion attacks are also preventable through resource limits. Since each flow setup request is attributable to a user, host, and access point, the Controller can enforce limits on the number of outstanding flows per identifiable source. The network may also support more advanced flow-allocation policies. For example, an integrated hardware/software Switch can implement policies such as enforcing strict limits on the number of flows forwarded in hardware per source and looser limits on the number of flows in the slower (and more abundant) software forwarding tables.

3.4 Handling Broadcast and Multicast

Enterprise networks typically carry a lot of multicast and broadcast traffic. It is worth distinguishing broadcast traffic (which is mostly discovery protocols, such as ARP) from multicast traffic (which is often from useful applications, such as video). In a flow-based network like Ethane, it is quite easy for Switches to handle multicast: The Switch keeps a bitmap for each flow to indicate which ports the packets are to be sent to along the path. The Controller can calculate the broadcast or multicast tree and assign the appropriate bits during path setup.

In principle, broadcast discovery protocols are also easy to handle in the Controller. Typically, a host is trying to find a server or an address; given that the Controller knows all, it can reply to a request without creating a new flow and broadcasting the traffic. This provides an easy solution for ARP traffic, which is a significant fraction of all network traffic In practice, however, ARP could generate a huge load for the Controller; one design choice would be to provide a dedicated ARP server in the network to which all Switches direct all ARP traffic. But there is a dilemma when trying to support other discovery protocols: each one has its own protocol, and it would be onerous for the Controller to understand all of them. Our own approach has been to implement the common ones directly in the Controller and to broadcast unknown request types with a rate-limit. Clearly this approach does not scale well, and we hope that, if Ethane becomes widespread in the future, discovery protocols will largely go away. After all, they are just looking for binding information that the network already knows; it should be possible to provide a direct way to query the network. We discuss this problem further in §7.

3.5 Replicating the Controller: Fault-Tolerance and Scalability

Designing a network architecture around a central controller raises concerns about availability and scalability. While our measurements in §6 suggest that thousands of machines can be managed by a single desktop computer, multiple Controllers may be desirable to provide fault-tolerance or to scale to very large networks.

This section describes three techniques for replicating the Controller. In the simplest two approaches, which focus solely on improving fault-tolerance, secondary Controllers are ready to step in upon the primary's failure: these can be in *cold-standby* (having no network binding state) or *warm-standby* (having network binding state) modes. In the *fully-replicated* model, which also improves scalability, requests from Switches are spread over multiple active Controllers.

In the cold-standby approach, a primary Controller is the root of the minimum spanning tree (MST) and handles all registration, authentication, and flow-establishment requests. Backup Controllers sit idly-by waiting to take over if needed. All Controllers participate in the MST, sending HELLO messages to Switches advertising their ID. Just as with a standard spanning tree, if the root with the "lowest" ID fails, the network will converge on a new root (*i.e.*, a new Controller). If a backup becomes the new MST root, it will start to receive flow requests and begin acting as the primary Controller. In this way, Controllers can be largely unaware of each other: the backups need only contain the registration state and the network policy (as this data changes very slowly, simple consistency methods can be used). The main advantage of cold-standby is its simplicity; the downside is that hosts, Switches, and users need to re-authenticate and re-bind upon the primary's failure. Furthermore, in large networks, it might take a while for the MST to reconverge.

The warm-standby approach is more complex, but recovers faster. In this approach, a separate MST is created for every Controller. The Controllers monitor one another's liveness and, upon detecting the primary's failure, a secondary Controller takes over based on a static ordering. As before, slowly-changing registration and network policy are kept consistent among the Controllers, but we now need to replicate bindings across Controllers as well. Because these bindings can change quickly as new users and hosts come and go, we recommend that only weak consistency be maintained: Because Controllers make bind events atomic, primary failures can at worst lose the latest bindings, requiring that some new users and hosts reauthenticate themselves.

The fully-replicated approach takes this one step further and has two or more active Controllers. While an MST is again constructed for each Controller, a Switch need only authenticate itself to one Controller and can then spread its flow-requests over the Controllers (*e.g.*, by hashing or round-robin). With such replication, we should not underestimate the job of maintaining consistent journals of the bind events. We expect that most implementations will simply use gossiping to provide a weakly-consistent ordering over events. Pragmatic techniques can avoid many potential problems that would otherwise arise, *e.g.*, having Controllers use different private IP address spaces during DHCP allocation to prevent temporary IP allocation conflicts. Of course, there are well-known, albeit heavier-weight, alternatives to provide stronger consistency guarantees if desired (*e.g.*, replicated state machines). There is plenty of scope for further study: Now that Ethane provides a platform with which to capture and manage all bindings, we expect future improvements can make the system more robust.

3.6 Link Failures

Link and Switch failures must not bring down the network as well. Recall that Switches always send neighbor-discovery messages to keep track of link-state. When a link fails, the Switch removes all flow table entries tied to the failed port and sends its new link-state information to the Controller. This way, the Controller also learns the new topology. When packets arrive for a removed flow-entry at the Switch, the packets are sent to the Controller—much like they are for new flows—and the Controller computes and installs a new path based on the new topology.

3.7 Bootstrapping

When the network starts, the Switches must connect to and authenticate with the Controller.[8] Ethane bootstraps in a similar way to SANE [12]: On startup, the network creates a minimum spanning tree with the Controller advertising itself as the root. Each

[8]This method does not apply to Switches that use an IP tunnel to connect to the Controller—they simply send packets via the tunnel and then authenticate.

Switch has been configured with the Controller's credentials and the Controller with the Switches' credentials.

If a Switch finds a shorter path to the Controller, it attempts two-way authentication with it before advertising that path as a valid route. Therefore, the minimum spanning tree grows radially from the Controller, hop-by-hop as each Switch authenticates.

Authentication is done using the preconfigured credentials to ensure that a misbehaving node cannot masquerade as the Controller or another Switch. If authentication is successful, the Switch creates an encrypted connection with the Controller that is used for all communication between the pair.

By design, the Controller knows the upstream Switch and physical port to which each authenticating Switch is attached. After a Switch authenticates and establishes a secure channel to the Controller, it forwards all packets it receives for which it does not have a flow entry to the Controller, annotated with the ingress port. This includes the traffic of authenticating Switches.

Therefore, the Controller can pinpoint the attachment point to the spanning tree of all non-authenticated Switches and hosts. Once a Switch authenticates, the Controller will establish a flow in the network between itself and the Switch for the secure channel.

4. THE *POL-ETH* POLICY LANGUAGE

Pol-Eth is a language for declaring policy in an Ethane network. While Ethane doesn't mandate a particular language, we describe *Pol-Eth* as an example, to illustrate what's possible. We have implemented *Pol-Eth* and use it in our prototype network.

4.1 Overview

In *Pol-Eth*, network policy is declared as a set of rules, each consisting of a *condition* and a corresponding *action*. For example, the rule to specify that user *bob* is allowed to communicate with the web server (using HTTP) is the following:

[(usrc="bob")∧(protocol="http")∧(hdst="websrv")]:allow;

Conditions. Conditions are a conjunction of zero or more predicates which specify the properties a flow must have in order for the action to be applied. From the preceding example rule, if the user initiating the flow is "bob" **and** the flow protocol is "HTTP" **and** the flow destination is host "websrv," then the flow is *allowed*. The left hand side of a predicate specifies the domain, and the right hand side gives the entities to which it applies. For example, the predicate *(usrc="bob")* applies to all flows in which the source is user *bob*. Valid domains include {*usrc, udst, hsrc, hdst, apsrc, apdst, protocol*}, which respectively signify the user, host, and access point sources and destinations and the protocol of the flow.

In *Pol-Eth*, the values of predicates may include single names (*e.g.*, "bob"), list of names (*e.g.*, ["bob","linda"]), or group inclusion (*e.g.*, in("workstations")). All names must be registered with the Controller or declared as groups in the policy file, as described below.

Actions. *Actions* include *allow*, *deny*, *waypoints*, and *outbound-only* (for NAT-like security). Waypoint declarations include a list of entities to route the flow through, *e.g.*, **waypoints("ids","web-proxy")**.

4.2 Rule and Action Precedence

Pol-Eth rules are independent and don't contain an intrinsic ordering; thus, multiple rules with conflicting actions may be satisfied by the same flow. Conflicts are resolved by assigning priorities based on declaration order. If one rule precedes another in the policy file, it is assigned a higher priority.

```
# Groups —
desktops = ["griffin","roo"];
laptops = ["glaptop","rlaptop"];
phones = ["gphone","rphone"];
server = ["http_server","nfs_server"];
private = ["desktops","laptops"];
computers = ["private","server"];
students = ["bob","bill","pete"];
profs = ["plum"];
group = ["students","profs"];
waps = ["wap1","wap2"];
%%
# Rules —
[(hsrc=in("server")∧(hdst=in("private"))] : deny;
# Do not allow phones and private computers to communicate
[(hsrc=in("phones")∧(hdst=in("computers"))] : deny;
[(hsrc=in("computers")∧(hdst=in("phones"))] : deny;
# NAT-like protection for laptops
[(hsrc=in("laptops")] : outbound-only;
# No restrictions on desktops communicating with each other
[(hsrc=in("desktops")∧(hdst=in("desktops"))] : allow;
# For wireless, non-group members can use http through
# a proxy. Group members have unrestricted access.
[(apsrc=in("waps"))∧(user=in("group"))] :allow;
[(apsrc=in("waps"))∧(protocol="http)] : waypoints("http-proxy");
[(apsrc=in("waps"))] : deny;
[]: allow; # Default-on: by default allow flows
```

Figure 4: A sample policy file using *Pol-Eth*

Unfortunately, in today's multi-user operating systems, it is difficult from a network perspective to attribute outgoing traffic to a particular user. In Ethane, if multiple users are logged into the same machine (and not identifiable from within the network), Ethane applies the least restrictive action to each of the flows. This is an obvious relaxation of the security policy. To address this, we are exploring integration with trusted end-host operating systems to provide user-isolation and identification (for example, by providing each user with a virtual machine having a unique MAC).

4.3 Policy Example

Figure 4 contains a derivative of the policy which governs connectivity for our university deployment. *Pol-Eth* policy files consist of two parts—group declarations and rules—separated by a '%%' delimiter. In this policy, all flows which do not otherwise match a rule are permitted (by the last rule). Servers are not allowed to initiate connections to the rest of the network, providing protection similar to DMZs today. Phones and computers can never communicate. Laptops are protected from inbound flows (similar to the protection provided by NAT), while workstations can communicate with each other. Guest users from wireless access points may only use HTTP and must go through a web proxy, while authenticated users have no such restrictions.

4.4 Implementation

Given how frequently new flows are created—and how fast decisions must be made—it is not practical to interpret the network policy. Instead, we need to compile it. But compiling *Pol-Eth* is non-trivial because of the potentially huge namespace in the network: Creating a lookup table for all possible flows specified in the policy would be impractical.

Our *Pol-Eth* implementation combines compilation and just-in-time creation of search functions. Each rule is associated with the principles to which it applies. This is a one-time cost, performed at startup and on each policy change.

The first time a sender communicates with a new receiver, a custom permission check function is created dynamically to handle all subsequent flows between this source/destination pair. The function is generated from the set of rules which apply to the connection. In the worst case, the cost of generating the function scales linearly with the number of rules (assuming each rule applies to every source entity). If all of the rules contain conditions that can be checked statically at bind time (*i.e.*, the predicates are defined only over users, hosts, and access points), then the resulting function consists solely of an *action*. Otherwise, each flow request requires that the actions be aggregated in real-time.

We have implemented a source-to-source compiler that generates C++ from a *Pol-Eth* policy file. The resulting source is then compiled and linked into the Ethane binary. As a consequence, policy changes currently require relinking the Controller. We are currently upgrading the policy compiler so that policy changes can be dynamically loaded at runtime.

5. PROTOTYPE AND DEPLOYMENT

We've built and deployed a functional Ethane network at our university over the last four months. At the time of writing, Ethane connects over 300 registered hosts and several hundred users. Our deployment includes 19 Switches of three different types: Ethane wireless access points and Ethane Ethernet switches in two flavors (one gigabit in dedicated hardware and one in software). Registered hosts include laptops, printers, VoIP phones, desktop workstations, and servers. We have also deployed a remote Switch in a private residence; the Switch tunnels to the remote Controller which manages all communications on the home network. The whole network is managed by a single PC-based Controller.

In the following section, we describe our Ethane prototype and its deployment within Stanford's Computer Science department, drawing some lessons and conclusions based on our experience.

5.1 Switches

We have built three different Ethane Switches: An 802.11g wireless access point (based on a commercial access point), a wired 4-port Gigabit Ethernet Switch that forwards packets at line-speed (based on the NetFPGA programmable switch platform [7] and written in Verilog), and a wired 4-port Ethernet Switch in Linux on a desktop PC (in software, both as a development environment and to allow rapid deployment and evolution).

For design re-use, we implemented the same flow table in each Switch design, even though in real-life we would optimize for each platform. The main table—for packets that should be *forwarded* (see Section 3.2)—has 8,192 flow entries and is searched using an exact match on the whole header. We use two hash functions (two CRCs) to reduce the chance of collisions, and we place only one flow in each entry of the table.

We implemented a second table with 32K entries. We did this because our Controller is stateless and we wanted to implement the *outbound-only* action in the flow table. When an outbound flow starts, we'd like to setup the return-route at the same time (because the Controller is stateless, it doesn't remember that the outbound-flow was allowed). Unfortunately, when proxy ARP is used, we don't know the MAC address of packets flowing in the reverse direction until they arrive. So, we use the second table to hold flow entries for return-routes (with a wildcard MAC address) as well as for dropped packets. A stateful Controller wouldn't need these entries.

Finally, we keep a small table for flows with wildcards in any field. This table is present for convenience during prototyping, while we determine how many entries a real deployment would need for bootstrapping and control traffic. It currently holds flow entries for the spanning tree, ARP, and DHCP messages.

Ethane Wireless Access Point. Our access point runs on a Linksys WRTSL54GS wireless router (266MHz MIPS, 32MB RAM) running OpenWRT [8]. The data-path and flow table are based on 5K lines of C++ (1.5K are for the flow table). The local switch manager is written in software and talks to the Controller using the native Linux TCP stack. When running from within the kernel, the Ethane forwarding path runs at 23Mb/s—the same speed as Linux IP forwarding and L2 bridging.

Ethane 4-port Gigabit Ethernet Switch: Hardware Solution. The Switch is implemented on NetFPGA v2.0 with four Gigabit Ethernet ports, a Xilinx Virtex-II FPGA, and 4MB of SRAM for packet buffers and the flow table. The hardware forwarding path consists of 7K lines of Verilog; flow entries are 40 bytes long. Our hardware can forward minimum-size packets in full-duplex at a line rate of 1Gb/s.

Ethane 4-port Gigabit Ethernet Switch: Software Solution. We also built a Switch from a regular desktop PC (1.6GHz Celeron CPU and 512MB of DRAM) and a 4-port Gigabit Ethernet card. The forwarding path and the flow table are implemented to mirror (and therefore help debug) our implementation in hardware. Our software Switch in kernel mode can forward MTU size packets at 1Gb/s. However, as the packet size drops, the switch can't keep pace with hashing and interrupt overheads. At 100 bytes, the switch can only achieve a throughput of 16Mb/s. Clearly, for now, the switch needs to be implemented in hardware for high-performance networks.

5.2 Controller

We implemented the Controller on a standard Linux PC (1.6GHz Celeron CPU and 512MB of DRAM). The Controller is based on 45K lines of C++ (with an additional 4K lines generated by the policy compiler) and 4.5K lines of Python for the management interface.

Registration. Switches and hosts are registered using a web interface to the Controller and the registry is maintained in a standard database. For Switches, the authentication method is determined during registration. Users are registered using our university's standard directory service.

Authentication. In our system, users authenticate using our university authentication system, which uses Kerberos and a university-wide registry of usernames and passwords. Users authenticate via a web interface—when they first connect to a browser they are redirected to a login web-page. In principle, any authentication scheme could be used, and most enterprises have their own. Ethane Switches also, optionally, authenticate hosts based on their MAC address, which is registered with the Controller.

Bind Journal and Namespace Interface. Our Controller logs bindings whenever they are added or removed, or when we decide to checkpoint the current bind-state; each entry in the log is timestamped. We use BerkeleyDB for the log [2], keyed by timestamp.

The log is easily queried to determine the bind-state at any time in the past. We enhanced our DNS server to support queries of the form *key.domain.type-time*, where "type" can be "host", "user", "MAC", or "port". The optional time parameter allows historical queries, defaulting to the present time.

Route Computation. Routes are pre-computed using an all pairs shortest path algorithm [13]. Topology recalculation on link failure is handled by dynamically updating the computation with the modified link-state updates. Even on large topologies, the cost of updating the routes on failure is minimal. For example, the aver-

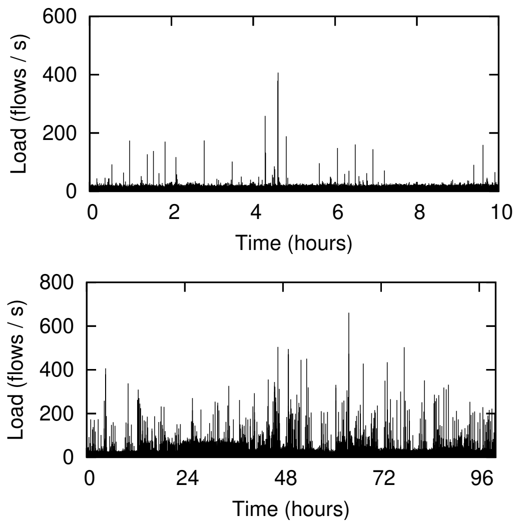

Figure 5: Frequency of flow-setup requests per second to Controller over a 10-hour period (top) and 4-day period (bottom).

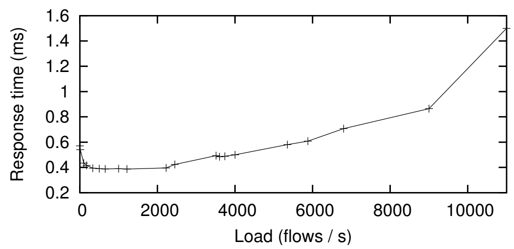

Figure 6: Flow-setup times as a function of Controller load. Packet sizes were 64B, 128B and 256B, evenly distributed.

age cost of an update on a 3,000 node topology is 10ms. In the following section we present an analysis of flow-setup times under normal operation and during link failure.

5.3 Deployment

Our Ethane prototype is deployed in our department's 100Mb/s Ethernet network. We installed eleven wired and eight wireless Ethane Switches. There are currently approximately 300 hosts on this Ethane network, with an average of 120 hosts active in a 5-minute window. We created a network policy to closely match—and in most cases exceed—the connectivity control already in place. We pieced together the existing policy by looking at the use of VLANs, end-host firewall configurations, NATs and router ACLs. We found that often the existing configuration files contained rules no longer relevant to the current state of the network, in which case they were not included in the Ethane policy.

Briefly, within our policy, non-servers (workstations, laptops, and phones) are protected from outbound connections from servers, while workstations can communicate uninhibited. Hosts that connect to an Ethane Switch port must register a MAC address, but require no user authentication. Wireless nodes protected by WPA and a password do not require user authentication, but if the host MAC address is not registered (in our network this means they are a guest), they can only access a small number of services (HTTP, HTTPS, DNS, SMTP, IMAP, POP, and SSH). Our open wireless access points require users to authenticate through the university-wide system. The VoIP phones are restricted from communicating with non-phones and are statically bound to a single access point to

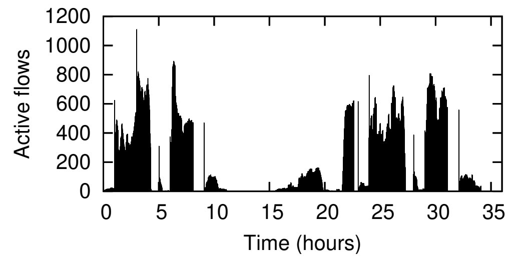

Figure 7: Active flows for LBL network [19].

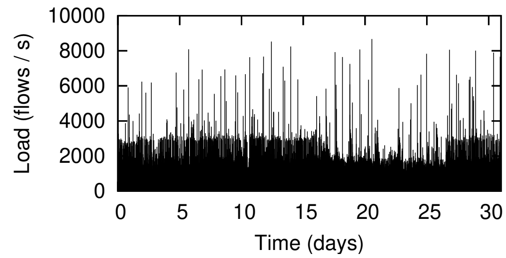

Figure 8: Flow-request rate for Stanford network.

prevent mobility (for E911 location compliance). Our policy file is 132 lines long.

6. PERFORMANCE AND SCALABILITY

Deploying Ethane has taught us a lot about the operation of a centrally-managed network, and it enabled us to evaluate many aspects of its performance and scalability, especially with respect to the numbers of users, end-hosts, and Switches. We start by looking at how Ethane performs in our network, and then, using our measurements and data from others, we try to extrapolate the performance for larger networks.

In this section, we first measure the Controller's performance as a function of the flow-request rate, and we then try to estimate how many flow-requests we can expect in a network of a given size. This allows us to answer our primary question: *How many Controllers are needed for a network of a given size?* We then examine the behavior of an Ethane network under Controller and link failures. Finally, to help decide the practicality and cost of Switches for larger networks, we consider the question: *How big does the flow table need to be in the Switch?*

6.1 Controller Scalability

Recall that our Ethane prototype is currently used by approximately 300 hosts, with an average of 120 hosts active in a 5-minute window. From these hosts, we see 30-40 new flow requests per second (Figure 5) with a peak of 750 flow requests per second.[9] Figure 6 shows how our Controller performs under load: for up to 11,000 flows per second—greater than the peak laod we observed—flows were set up in less than 1.5 milliseconds in the worst case, and the CPU showed negligible load.

Our results suggest that a single Controller could comfortably handle 10,000 new flow requests per second. We fully expect this number to increase if we concentrated on optimizing the design. With this in mind, it is worth asking to how many end-hosts this load corresponds.

We considered two recent datasets: One from an 8,000-host network at LBL [19] and one from a 22,000-host network at Stanford. As is described in [12], the number of maximum outstanding flows

[9]Samples were taken every 30 seconds.

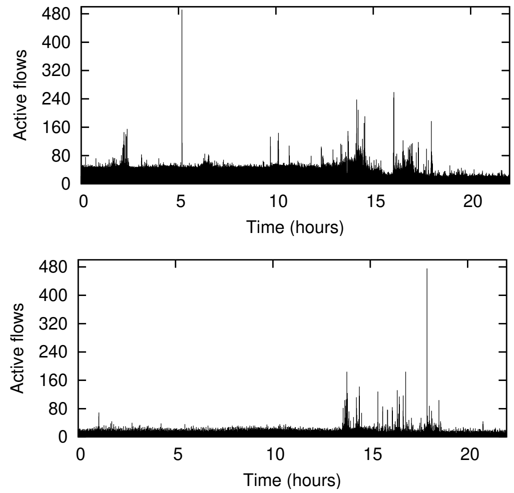

Figure 9: Active flows through two of our deployed switches

Failures	0	1	2	3	4
Completion time	26.17s	27.44s	30.45s	36.00s	43.09s

Table 1: Completion time for HTTP GETs of 275 files during which the primary Controller fails zero or more times. Results are averaged over 5 runs.

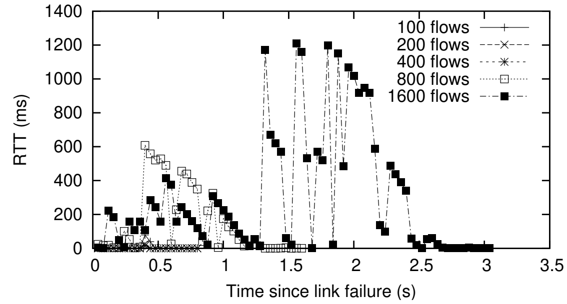

Figure 10: Round-trip latencies experienced by packets through a diamond topology during link failure.

in the traces from LBL never exceeded 1,200 per second across all nodes (Figure 7). The Stanford dataset has a maximum of under 9,000 new flow-requests per second (Figure 8).

Perhaps surprisingly, our results suggest that a single Controller could comfortably manage a network with over 20,000 hosts. Indeed flow setup latencies for continued load of up to 6,000/s are less than .6ms, equivalent to the average latency of a DNS request within the Stanford network. Flow setup latencies for load under 2,000 requests per second are .4ms, this is roughly equivalent to the average RTT between hosts in different subnets on our campus network.

Of course, in practice, the rule set would be larger and the number of physical entities greater. On the other hand, the ease with which the Controller handles this number of flows suggests there is room for improvement. This is not to suggest that a network should rely on a single Controller; we expect a large network to deploy several Controllers for fault-tolerance, using the schemes outlined in §3.5, one of which we examine next.

6.2 Performance During Failures

Because our Controller implements cold-standby failure recovery (see §3.5), a Controller failure will lead to interruption of service for active flows and a delay while they are re-established. To understand how long it takes to reinstall the flows, we measured the completion time of 275 consecutive HTTP requests, retrieving 63MB in total. While the requests were ongoing, we crashed the Controller and restarted it multiple times. Table 1 shows that there is clearly a penalty for each failure, corresponding to a roughly 10% increase in overall completion time. This can be largely eliminated, of course, in a network that uses warm-standby or fully-replicated Controllers to more quickly recover from failure (see §3.5).

Link failures in Ethane require that all outstanding flows re-contact the Controller in order to re-establish the path. If the link is heavily used, the Controller will receive a storm of requests, and its performance will degrade. We created a topology with redundant paths—so the network can withstand a link-failure—and measured the latencies experienced by packets. Failures were simulated by physically unplugging a link; our results are shown in Figure 10. In all cases, the path reconverges in under 40ms, but a packet could be delayed up to a second while the Controller handles the flurry of requests.

Our network policy allows for multiple disjoint paths to be setup by the Controller when the flow is created. This way, convergence can occur much faster during failure, particularly if the Switches detect a failure and failover to using the backup flow-entry. We have not implemented this in our prototype, but plan to do so in the future.

6.3 Flow Table Sizing

Finally, we explore how large the flow table needs to be in the Switch. Ideally, the Switch can hold all of the currently active flows. Figure 9 shows how many active flows we saw in our Ethane deployment; it never exceeded 500. With a table of 8,192 entries and a two-function hash-table, we never encountered a collision. As described earlier in Figure 7, the LBL network did not encounter more than 1,200 flows in their 8,000 host network.

In practice, the number of ongoing flows depends on where the Switch is in the network. Switches closer to the edge will see a number of flows proportional to the number of hosts they connect to (*i.e.*, their fanout). Our deployed Switches have a fanout of four and saw no more than 500 flows; we might expect a Switch with a fanout of, say, 64 to see at most a few thousand active flows. (It should be noted that this is a very conservative estimate, given the small number of flows in the whole LBL network.) A Switch at the center of a network will likely see more active flows, and so we assume it will see all active flows.

From these numbers we conclude that a Switch—for a university-sized network—should have flow table capable of holding 8K–16K entries. If we assume that each entry is 64B, such a table requires about 1MB of storage, or as much as 4MB if we use a two-way hashing scheme [9]. A typical commercial enterprise Ethernet switch today holds 1 million Ethernet addresses (6MB, but larger if hashing is used), 1 million IP addresses (4MB of TCAM), 1-2 million counters (8MB of fast SRAM), and several thousand ACLs (more TCAM). Thus, the memory requirements of an Ethane Switch are quite modest in comparison to today's Ethernet switches.

To further explore the scalability of the Controller, we tested its performance with simulated inputs in software to identify overheads. The Controller was configured with a policy file of 50 rules and 100 registered principles; routes were precalculated and cached. Under these conditions, the system could handle 650,845 bind events per second and 16,972,600 permission checks per second. The complexity of the bind events and permission checks is dependent on the rules in use, which, in the worst case, grows linearly with the number of rules.

7. ETHANE'S SHORTCOMINGS

When trying to deploy a radically new architecture into legacy networks—without changing the end-host—we encounter some stumbling blocks and limitations. These are the main issues that arose:

Broadcast and Service Discovery. Broadcast discovery protocols (ARP, OSPF neighbor discovery, etc.) wreak havoc on enterprise networks by generating huge amounts of overhead traffic [17, 20]; on our network, these constituted over 90% of the flows. One of the largest reasons for VLANs is to control the storms of broadcast traffic on enterprise networks. Hosts frequently broadcast messages to the network to try and find an address, neighbor, or service. Unless Ethane can interpret the protocol and respond on its behalf, it needs to broadcast the request to all potential responders; this involves creating large numbers of flow entries, and it leads to lots of traffic which—if malicious—has access to every end-host. Broadcast discovery protocols could be eliminated if there was a standard way to register a service where it can easily be found. SANE proposed such a scheme [12], and in the long-term, we believe this is the right approach.

Application-layer routing. A limitation of Ethane is that it has to trust end-hosts not to relay traffic in violation of the network policy. Ethane controls connectivity using the Ethernet and IP addresses of the end-points, but Ethane's policy can be compromised by communications at a higher layer. For example, if *A* is allowed to talk to *B* but not *C*, and if *B* can talk to *C*, then *B* can relay messages from *A* to *C*. This could happen at any layer above the IP layer, *e.g.*, a P2P application that creates an overlay at the application layer, or multi-homed clients that connect to multiple networks. This is a hard problem to solve, and most likely requires a change to the operating system and any virtual machines running on the host.

Knowing what the user is doing. Ethane's policy assumes that the transport port numbers indicate what the user is doing: port 80 means HTTP, port 25 is SMTP, and so on. Colluding malicious users or applications can fool Ethane by agreeing to use non-standard port numbers. And it is common for "good" applications to tunnel applications over ports (such as port 80) that are likely to be open in firewalls. To some extent, there will always be such problems for a mechanism like Ethane, which focuses on connectivity without involvement from the end-host. In the short-term, we can, and do, insert application proxies along the path (using Ethane's waypoint mechanism).

Spoofing Ethernet addresses. Ethane Switches rely on the binding between a user and Ethernet addresses to identify flows. If a user spoofs a MAC address, it might be possible to fool Ethane into delivering packets to an end-host. This is easily prevented in an Ethane-only network where each Switch port is connected to one host: The Switch can drop packets with the wrong MAC address. If two or more end-hosts connect to the same Switch port, it is possible for one to masquerade as another. A simple solution is to physically prevent this; a more practical solution in larger networks is to use 802.1X in conjunction with link-level encryption mechanisms, such as 802.1AE, to more securely authenticate packets and addresses.

8. RELATED WORK

Ethane embraces the 4D [14] philosophy of simplifying the data-plane and centralizing the control-plane to enforce network-wide goals [21]. Ethane diverges from 4D in that it supports a fine-grained policy-management system. We believe that policy decisions can and should be based on flows. We also believe that by moving all flow decisions to the Controller, we can add many new functions and features to the network by simply updating the Controller in a single location. Our work also shows that it is possible—we believe for the first time—to securely bind the entities in the network to their addresses, and then to manage the whole namespace with a single policy.

Ipsilon Networks proposed caching IP routing decisions as flows, in order to provide a switched, multi-service fast path to traditional IP routers [18]. Ethane also uses flows as a forwarding primitive. However, Ethane extends forwarding to include functionality useful for enforcing security, such as address swapping and enforcing outgoing initiated flows only.

In distributed firewalls [15], policy is declared centrally in a topology independent manner and enforced at each end-host. In addition to the auditing and management support, Ethane differs from this work in two major ways. First, in Ethane end-hosts cannot be trusted to enforce filtering. This mistrust is also extended to the first hop switch. With per-switch enforcement of each flow, Ethane provides maximal defense in depth. Secondly, much of the power of Ethane is to provide network level guarantees, such as policy imposed waypoints. This is not possible to do through end-host level filtering alone.

Pol-Eth, Ethane's policy language, is inspired by predicate routing (PR) [22]. PR unifies routing and filtering; a set of predicates describes all connectivity. *Pol-Eth* extends this model by making users first-class objects, declaring predicates over high-level names, and providing support for group declaration and inclusion, multiple connectivity constraints, and arbitrary expressions.

VLANs are widely used in enterprise networks for segmentation, isolation, and to enforce coarse-grain policies; and they are commonly used to quarantine unauthenticated hosts or hosts without health "certificates" [3, 6]. VLANs are notoriously difficult to use, requiring much hand-holding and manual configuration; we believe Ethane can replace VLANs entirely, giving much simpler control over isolation, connectivity, and diagnostics.

There are a number of Identity-Based Networking (IBN) custom switches available (*e.g.*, [4]) or secure AAA servers (*e.g.*, [5]). These allow high-level policy to be declared, but are generally point solutions with little or no control over the network data-path (except as a choke-point). Several of them rely on the end-host for enforcement, which makes them vulnerable to compromise.

9. CONCLUSIONS

One of the most interesting consequences of building a prototype is that the lessons you learn are always different—and usually far more—than were expected. With Ethane, this is most definitely the case: We learned lessons about the good and bad properties of Ethane, and fought a number of fires during our deployment.

The largest conclusion that we draw is that (once deployed) we found it much easier to manage the Ethane network than we expected. On numerous occasions we needed to add new Switches, new users, support new protocols, and prevent certain connectivity. On each occasion we found it natural and fast to add new policy rules in a single location. There is great peace of mind to knowing that the policy is implemented at the place of entry and determines the route that packets take (rather than being distributed as a set of filters without knowing the paths that packets follow). By journaling all registrations and bindings, we were able to identify numerous network problems, errant machines, and malicious flows—and associate them with an end-host or user. This bodes well for network managers who want to hold users accountable for their traffic or perform network audits.

We have also found it straightforward to add new features to the network: either by extending the policy language, adding new routing algorithms (such as supporting redundant disjoint paths), or introducing new application proxies as waypoints. Overall, we believe that Ethane's most significant advantage comes from the ease of innovation and evolution. By keeping the Switches dumb and simple, and by allowing new features to be added in software on the central Controller, rapid improvements are possible. This is particularly true if the protocol between the Switch and Controller is open and standardized, so as to allow competing Controller software to be developed.

We are confident that the Controller can scale to support quite large networks: Our results suggest that a single Controller could manage over 10,000 machines, which bodes well for whoever has to manage the Controllers. In practice, we expect Controllers to be replicated in topologically-diverse locations on the network, yet Ethane does not restrict how the network manager does this. Over time, we expect innovation in how fault-tolerance is performed, perhaps with emerging standard protocols for Controllers to communicate and remain consistent.

We are convinced that the Switches are best when they are dumb, and contain little or no management software. We have experience building switches and routers—for Ethane and elsewhere—and these are the simplest switches we've seen. Further, the Switches are just as simple at the center of the network as they are at the edge. Because the Switch consists mostly of a flow table, it is easy to build in a variety of ways: in software for embedded devices, in network processors, for rapid deployment, and in custom ASICs for high volume and low-cost. Our results suggest that an Ethane Switch will be significantly simpler, smaller, and lower-power than current Ethernet switches and routers.

We anticipate some innovation in Switches too. For example, while our Switches maintain a single FIFO queue, one can imagine a "less-dumb" Switch with multiple queues, where the Controller decides to which queue a flow belongs. This leads to many possibilities: per-class or per-flow queueing in support of priorities, traffic isolation, and rate control. Our results suggest that even if the Switch does per-flow queueing (which may or may not make sense), the Switch need only maintain a few thousand queues. This is frequently done in low-end switches today, and it is well within reach of current technology.

Acknowledgments

We thank Tal Garfinkel, Greg Watson, Dan Boneh, Lew Glendenning, John Lockwood, and Aditya Akella for their valuable input. David Mazières, Nickolai Zeldovich, Jennifer Rexford, Sharon Goldberg and Changoon Kim offered very helpful feedback on early drafts of this paper. We would also like to thank our shepherd, Anja Feldman for her guidance. This paper is based upon work supported by the National Science Foundation under Grant No. CNS-0627112 (The 100x100 Clean Slate Program), from the FIND program with funding from DTO. The research was also supported in part by the Stanford Clean Slate program. Martìn Casado was funded by a DHS graduate fellowship.

References

[1] Alterpoint. http://www.alterpoint.com/.

[2] BerkeleyDB. http://www.oracle.com/database/berkeley-db.html.

[3] Cisco network admission control. http://www.cisco.com/.

[4] Consentry. http://www.consentry.com/.

[5] Identity engines. http://www.idengines.com/.

[6] Microsoft network access protection. http://www.microsoft.com/technet/network/nap/default.mspx.

[7] Netfpga home page. http://NetFPGA.org.

[8] Openwrt home page. http://openwrt.org/.

[9] A. Z. Broder and M. Mitzenmacher. Using multiple hash functions to improve ip lookups. In *Proc. INFOCOM*, Apr. 2001.

[10] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford. The cutting edge of ip router configuration. *SIGCOMM Computer Comm. Rev.*, 2004.

[11] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford. The cutting edge of ip router configuration. *SIGCOMM Computer Comm. Rev.*, 34(1):21–26, 2004.

[12] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A protection architecture for enterprise networks. In *USENIX Security Symposium*, Aug. 2006.

[13] C. Demetrescu and G. Italiano. A new approach to dynamic all pairs shortest paths. In *Proc. STOC'03*, 2003.

[14] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. In *SIGCOMM Computer Comm. Rev.*, Oct. 2005.

[15] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *ACM Conference on Computer and Communications Security*, pages 190–199, 2000.

[16] Z. Kerravala. Configuration management delivers business resiliency. *The Yankee Group*, Nov. 2002.

[17] A. Myers, E. Ng, and H. Zhang. Rethinking the service model: Scaling ethernet to a million nodes. In *Proc. HotNets*, Nov. 2004.

[18] P. Newman, T. L. Lyon, and G. Minshall. Flow labelled IP: A connectionless approach to ATM. In *INFOCOM (3)*, 1996.

[19] 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*, Oct. 2005.

[20] R. J. Perlman. Rbridges: Transparent routing. In *Proc. INFOCOM*, Mar. 2004.

[21] J. Rexford, A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, G. Xie, J. Zhan, and H. Zhang. Network-wide decision making: Toward a wafer-thin control plane. In *Proc. HotNets*, Nov. 2004.

[22] T. Roscoe, S. Hand, R. Isaacs, R. Mortier, and P. Jardetzky. Predicate routing: Enabling controlled networking. *SIGCOMM Computer Comm. Rev.*, 33(1), 2003.

[23] A. Wool. The use and usability of direction-based filtering in firewalls. *Computers & Security*, 26(6):459–468, 2004.

[24] A. Wool. A quantitative study of firewall configuration errors. *IEEE Computer*, 37(6):62–67, 2004.

[25] G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, and G. Hjalmtysson. Routing design in operational networks: A look from the inside. In *Proc. SIGCOMM*, Sept. 2004.