Rethinking Enterprise Network Control

Martín Casado, *Student Member, IEEE*, Michael J. Freedman, *Member, IEEE*, Justin Pettit, Jianying Luo, Natasha Gude, Nick McKeown, *Fellow, IEEE*, and Scott Shenker, *Fellow, IEEE*

***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. We also show that it is compatible with existing high-fanout switches by porting it to popular commodity switching chipsets. We have deployed and managed two operational Ethane networks, one in the Stanford University Computer Science Department supporting over 300 hosts, and another within a small business of 30 hosts. Our deployment experiences have significantly affected Ethane's design.**

***Index Terms*—Architecture, management, network, security.**

I. 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 [1]–[4] to achieve even moderate security [5]. A Yankee Group report found that 62% of network downtime in multivendor networks comes from human error and that 80% of IT budgets is spent on maintenance and operations [6].

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 [3]. 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 [7], [8] that help automate configuration management in order to reduce the risk of errors. However, these solutions hide the complexity and do not reduce it. Also, 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.

Network routing should be policy-aware. Network policies dictate the nature of connectivity between communicating entities and, therefore, naturally affect the paths that packets take. This is in contrast to today's networks in which forwarding and filtering use different mechanisms rather than a single integrated approach.

A 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. Policy may also specify service priorities for different classes of traffic. 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.

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 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

Manuscript received May 19, 2008; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor N. Taft. First published July 21, 2009; current version published August 19, 2009. This work was supported in part by the National Science Foundation under Grant CNS-0627112 (The 100 × 100 Clean Slate Program), by the FIND Program with funding from DTO, and by the Stanford Clean Slate Program. M. Casado was supported by a DHS graduate fellowship.

M. Casado, J. Pettit, J. Luo, N. Gude, and N. McKeown are with Stanford University, Stanford, CA 94305 USA (e-mail: casado@cs.stanford.edu; jpettit@cs.stanford.edu; jyluo@stanford.edu; ngude@cs.stanford.edu; nickm@stanford.edu).

M. J. Freedman is with Princeton University, Princeton, NJ 08544 USA (e-mail: mfreed@cs.princeton.edu).

S. Shenker is with the University of California, Berkeley, Berkeley, CA 94720 USA (e-mail: shenker@icsi.berkeley.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2009.2026415

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 [9] 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 variable, 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., thousands of hosts) from a single commodity PC.

Ethane bears substantial resemblance to SANE, our recently proposed clean-slate approach to enterprise security [10]. 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.

1) *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.
2) *Incremental deployability*: SANE required a "forklift" 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.
3) *Significant deployment experience*: Ethane has been implemented in both software and hardware (using special-purpose Gigabit Ethernet switches and commodity switch-on-a-chip products) and deployed at Stanford University's Computer Science Department for over four months and managed over 300 hosts. We also have experience deploying Ethane in a small business of over 30 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 Section II, followed by a detailed description in Section III. In Section IV, we describe a policy language *FSL* that we use to manage our Ethane implementation. We then discuss our implementation and deployment experience (Section V), followed by performance analysis (Section VI). Finally, we present limitations (Section VII), discuss related work (Section VIII), and then conclude (Section IX).

II. 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, the Switch forwards 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.

A. *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

[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."

network where there are no widely used protocols for keeping this information consistent. Furthermore, 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 example, 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.[3]

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.

A challenging component of the Ethane design was to create a policy language that operates over high-level names, is sufficiently expressive, and is fast enough to support a large network. Our solution was to design a new language called the Flow-based Security Language (*FSL*) [11], which is based on a restricted form of DATALOG. *FSL* policies are composed of sets of rules describing which flows they pertain to (via a conjunction of predicates) and the action to perform on those flows (e.g., allow, deny, or route via a waypoint). As we will see, *FSL*'s small set of easily understood rules can still express powerful and flexible policies, and it is possible to implement with performance suitable for very large networks.

B. Ethane in Use

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

a) *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).

b) *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.

c) *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,[4] 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.[5] Once authenticated, user_A is bound to host_A.

d) *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.

e) *Forwarding*:
   1) If the Controller allowed the path, it sends the packet back to switch 1, which forwards it based on the new

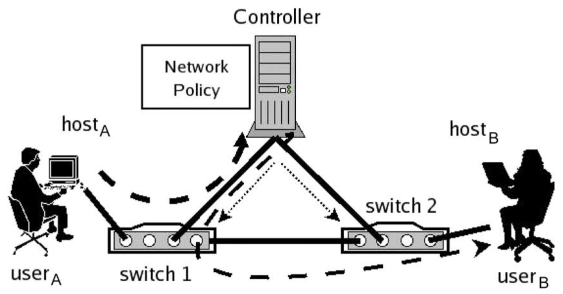

Fig. 1. Example of communication on an Ethane network. Route setup is shown by dotted lines; the path taken by the first packet of a flow is shown by dashed lines.

[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.

[3]We discuss the policy and security issues (and possible future solutions) with multiuser machines later in Sections IV and VII.

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

[5]Alternative authentication strategies may also be employed, e.g., 802.1X.

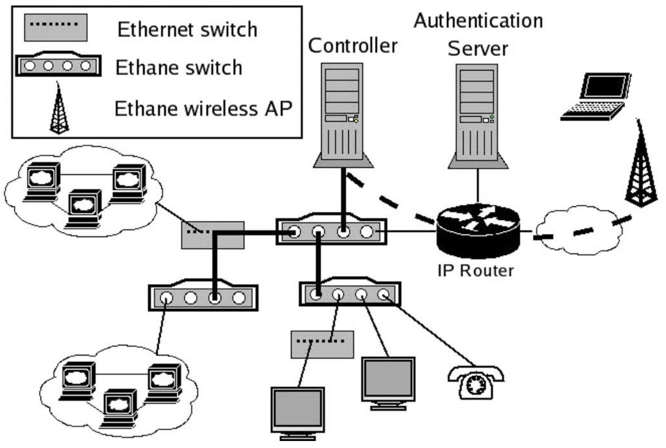

Fig. 2. An example Ethane deployment.

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.

III. Ethane in More Detail

A. Ethane Network

Fig. 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 Section III-B).[6]

When we add an Ethane Switch to the network, it has to find the Controller (Section III-C), 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 Section III-G 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).

B. 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.[7] 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.[8] 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

[6]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 it is to manage and the more secure the network.

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

[8]If a host is spoofing, its first-hop port can be shut off directly (Section III-C).

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.

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 that Switches 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 Section III-G. 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 s.

C. Controller

The Controller is the brain of the network and has many tasks; Fig. 3 gives a block-diagram. The components do not have to be colocated 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 Section IV). When a new flow starts, it is checked against the rules to see if it should be accepted, denied,

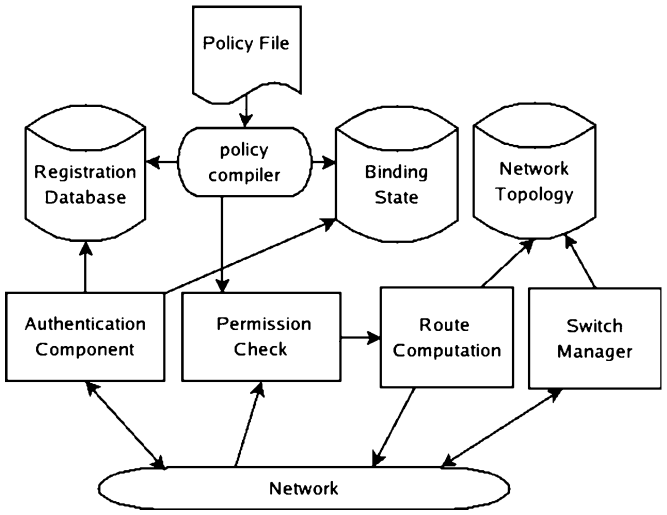

Fig. 3. High-level view of Controller components.

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[9]) must be registered. The set of registered entities makes up the policy namespace and is used to statically check the policy (Section IV) 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

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

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 Section V).

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 that determines the path given the policy constraint. In our implementation, all paths are precomputed and maintained via a dynamic all-pairs shortest-path algorithm [12]. Section IV describes our policy model and implementation.

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 flow entry that limits the rate of the offending packets or drops them entirely.

The Controller's ability to directly manage resources 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 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.

D. 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 Section VII.

E. 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 Section VI 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.

Controller distribution in Ethane takes advantage of two properties of its workload: 1) per-flow permission checks can be handled in a purely distributed fashion under the assumption that the bindings and network topology is the same on all replicas; and 2) changes to the network state happen on very slow time scales relative to flow arrivals (three to four orders of magnitude difference in our experience). Thus, the replication strategy is to load balance flows among the controller replicas[10] and ensure that the bindings and the network topology remain strongly consistent using a standard consensus protocol such as Paxos [13]. Only flows that result in a change of the controller state (such as a user or host authentication, or a link change) will incur overhead for consistency management.

When replicated, each Controller operates as the root of a spanning tree. Similar to the nondistributed case, Switches connect to all Controllers along the constructed MSTs. Controllers also use the MSTs to communicate with each other. If a Controller fails or becomes disconnected from a switch, the switch will no longer use the Controller's MST when load balancing flow permission checks. When a Controller joins the network, it will advertise a new spanning tree root which, after switch authentication, will be used to load balance flows.

The simplest form of replication requires that all Controllers be colocated and therefore no two can be disconnected by a network partition. However, this raises the possibility that a localized event (such as a power outage) could affect all Controllers on the network. On the other hand, while more resilient to failure, topologically distributing Controllers must be able to handle network partitions in which each partition is managed by a distinct set of Controllers. On network join, the Controllers must resolve their state, which will likely have diverged during the disconnection. Our current approach for dealing reconnection after a partition is for one of the disconnected Controller sets to drop all of their local changes and synchronize with the other. This may require some users or hosts to reauthenticate.

There is clearly plenty of scope in this area 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.

F. 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.

G. Bootstrapping

When the network starts, the Switches must connect to and authenticate with the Controller.[11] Ethane bootstraps in a similar way to SANE [10]: On startup, the network creates a minimum spanning tree with the Controller advertising itself as the root. Each 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 nonauthenticated 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.

IV. *FSL* Policy Language

The administrative interface to an Ethane network is the policy language. While the Ethane architecture is conceptually independent of the language, in order to be practical, the language must be designed with performance as a primary objective. To this end, we have developed a DATALOG-based language with negation called *FSL*. FSL supports distributed authorship, incremental updates, and automatic conflict resolution of network policy files.[12] We have implemented *FSL* and use it in one of our deployed networks.

A. Overview

Conceptually, an *FSL* policy is a set of functions mapping unidirectional flows to constraints that should be placed on those flows. *FSL* was designed so that given an Ethane flow, the desired constraints can be calculated efficiently.

In practice, an *FSL* 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

$$[\mathbf{allow}() <= \mathbf{usrc}(\text{“}\mathbf{bob}\text{"}) \wedge \mathbf{tpdst}(\text{“}\mathbf{http}\text{"}) \wedge \mathbf{hdst}(\text{“}\mathbf{websrv}\text{"})].$$

Conditions: A condition is a conjunction of zero or more literals describing the set of flows an action should be applied to. From the preceding example, if the user initiating the flow is "bob" **and** the flow destination transport protocol is "HTTP" **and** the flow destination is host "websrv," then the flow is *allowed*. In general, the predicate function (e.g., *usrc*) is the domain to be constrained, and the arguments are entities in that domain to which the rule applies. For example, the literal *usrc("bob")* applies to all flows in which the source user is *bob*.

[10]For example, by using consistent hashing over the source MAC address.

[11]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.

[12]Due to space constraints we only provide a brief introduction to *FSL*. A more detailed description can be found at [11].

Valid domains include {*apsrc, apdst, hsrc, hdst, usrc, udst, tpsrc, tpdst, protocol*}, which respectively signify the source and destination access point, host, user, and transport protocol of a flow, followed by the flow's protocol, where the argument should be a defined constant describing a subset of a flow's Ethernet, network, and transport protocols. Flows can also be described by the boolean predicates *isConnRequest()* and *isConnResponse()* to encode the role of a unidirectional flow in the larger connection setting. These can be used to restrict hosts to either outbound or inbound only connections, providing NAT-like security in the former case, and restricting servers from making outbound connections in the latter.

In *FSL*, predicate arguments can be single names (e.g., "*bob*") or lists of names (e.g., *["bob," "linda"]*). Literals can also be written as group inclusion expressions (e.g., *in("workstations", HSRC)*). Names may be registered statically at the Controller or can be resolved through a third-party authentication store such as LDAP or AD.

Actions: The standard set of *actions* include *allow*, *deny*, and *waypoint*. Waypoint declarations include a list of entities to route the flow through, e.g., *waypoint("ids", "Web-proxy")*.

Our *FSL* implementation allows the action set to be extended to arbitrary functions written in C++ or Python. This enables the very natural interposition of new services on the processing path. For example, we use this feature to define authentication policies by implementing an action *http_redirect*, which is applied to all unauthenticated packets. The function redirects users to a captive Web portal through which the user logs in.

B. Rule and Action Precedence

FSL rules are independent and do not contain an intrinsic ordering; thus, a single flow may match multiple rules with conflicting actions. Conflicts are resolved in two ways. The author can resolve them statically by assigning priorities using a cascading mechanism. This allows an administrator to quickly relax a security policy by inserting a high prioirity exception without having to understand the full policy file. Conflicts arising at runtime (not always detectable at compile time since literal values such as group membership can change dynamically), are resolved by selecting the *most secure* action. For example, *deny* is more secure than *waypoint*, which in turn is more secure than *allow*.

Unfortunately, in today's multiuser operating systems, it is difficult from a network perspective to attribute traffic to a particular user. In Ethane, if multiple users are logged into the same machine (and are not differentiable based on network identifiers), the least restrictive action present among the perhaps differently privileged users is applied to the machine's 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 with a unique MAC).

C. Policy Example

Fig. 4 lists a (prioritized) set of rules derived from the practices in one of our deployments. In this policy, all flows that do not otherwise match a rule are denied (by the last rule), enforcing a network security policy that can loosely be described as "default off." Unauthenticated users on laptops are sent to a captive Web portal for authentication. Servers and printers are not allowed to initiate connections. Laptops and mobile devices are protected from inbound flows (similar to the protection provided by NAT), while workstations can communicate without constraints.

```
allow() <= tpdst(8888) ∧ hdst("emerson")
fn_action("http_redirect") <= in('laptops',HSRC) ∧ usrc("unauthenticated")
# allow ARP and DHCP
allow() <= protocol('arp')
allow() <= protocol('dhcps') ∧ hdst("gateway")
allow() <= protocol('dhcpc') ∧ hsrc("gateway")
# allow computers to ssh into anyone
allow() <= protocol('ssh') ∧ in('computers', HSRC)
# dissallow testing machines from communicating externally
deny() <= in('testing', HSRC) | in('testing', HDST)
# servers should be inbound-only
deny() <= isConnRequest() ∧ (in('servers', HSRC) | in('printers', HSRC))
# printers should be inbound-only
deny() <= isConnRequest() ∧ (in('printers', HSRC) | in('printers', HSRC))
# laptops and mobile devices should be outbound-only
deny() <= isConnRequest() ∧ (in('mobile', HDST) | in('laptops', HDST))
# allow workstations unfettered access
allow() <= in('workstations', HSRC) | in('workstations', HDST)
# allow known devices to communicate as long as they abide by the
# previous rules.
allow() <= in('all', HSRC)
# default deny
deny() <= True
```

Fig. 4. A sample policy file using *FSL*.

D. Implementation

We have implemented *FSL* within Ethane. The compiler is written in Python and generates low-level C++ lookup structures traversed at runtime. Our implementation additionally permits dynamic changes to both the policy as well as group definitions.

In benchmarks using generated traffic, our implementation running our internal policy file supports permission checks for over 90 000 flows/s. As we discuss in the following section, this is more than adequate for the networks we've measured. Furthermore, our implementation maintains a relatively modest memory footprint even with large policy files: A 10 000 rule file uses less than 57 MB.

V. Prototype and Deployment

We have built and deployed two functional Ethane networks. Our initial deployment was at Stanford University, in which Ethane connected over 300 registered hosts and several hundred users. That deployment included 19 Switches, both wired and wireless. Our second deployment was in a small business network managing over 30 hosts. Both deployments are in operational networks entrusting Ethane to handle production traffic.

In the following section, we describe our Ethane prototypes and some aspects of their deployment, drawing some lessons and conclusions based on our experience.

A. Switches

We have built four different Ethane Switches from the ground up: 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 [14] and written in Verilog), a wired 16-port Ethane Switch running on a network appliance, and a wired 4-port Ethernet Switch in Linux on a desktop PC (in software, both as a development environment and to allow rapid prototyping).

In addition, we have ported Ethane to run on commodity switch platforms. When porting to switching hardware architectures, we are constrained by the existing hardware design. For example, in our experience, ACL rule engines in many chips can be used for flow lookup, however the number of rules (or flows in our case) supported is low. In one of our ports, the switch only allows 4096 rules for 48 ports of gigabit Ethernet. In such cases, we also implement a full software solution on the management processors to handle overflow, however the disparity in processing power makes this an inadequate approach for production use. Fortunately, there are now chips available that can leverage external memory to support tens of thousands of flows.

We now discuss each prototype Switch implementation in more detail.

- **Ethane Wireless Access Point.** Our access point runs on a Linksys WRTSL54GS wireless router (266 MHz MIPS, 32 MB RAM) running OpenWRT [15]. The datapath 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 23 Mb/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 4 MB 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 1 Gb/s.
- **Ethane 4-Port Gigabit Ethernet Switch: Software Solution.** We also built a Switch from a regular desktop PC (1.6 GHz Celeron CPU and 512 MB 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 1 Gb/s.[13]
- **Ethane 14-Port Ethernet Switch.** We were able to run our Linux kernel module unmodified on a 14-port Portwell Kilin-6030. This network appliance is based on the 16-core Cavium Octeon CN3860 processor. The advantage of this platform is that standard DRAM memory is used for flow lookup and, thus, can support an arbitrary number of flows. It is obviously faster than our PC-based solution, but we have not been able to measure its full capacity.
- **Ethane 48-Port Ethernet Switch.** We ported Ethane to a commercial 48-port switch running a pair of Broadcom BCM56514 chips. To do so, we modified the firmware to run the Ethane Linux kernel module on the on-board 1-GHz management CPU. Instead of adding flows in software, the module first attempts to insert rules into the switching chips' ACL engines. When all 4096 rules are exhausted, Ethane will default to using software flow entries. As long as rules match in hardware, the switch is able to support all 48 ports in full-duplex at a line rate of 1 Gb/s.

We plan to continue to investigate other platforms and optimizations. A very promising avenue is the Broadcom 566xx series of chips, which support hundreds of thousands of entries. We are also looking into making better use of the existing flow entry space by employing intelligent replacement policies such as LRU.

B. *Controller*

We implemented the Controller on a standard Linux PC (1.6 GHz Celeron CPU and 512 MB 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.

1) *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.
2) *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 already have an existing authentication infrastructure. Ethane Switches also, optionally, authenticate hosts based on their MAC address, which is registered with the Controller.
3) *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 [16], 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.
4) *Route Computation*: Routes are precomputed using an all pairs shortest path algorithm [12]. 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 average cost of an update on

[13]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 16 Mb/s. Clearly, for now, the switch needs to be implemented in hardware for high-performance networks.

a 3000-node topology is 10 ms. In the following section, we present an analysis of flow-setup times under normal operation and during link failure.

C. University Deployment

We describe the larger of our two deployments as an example of an Ethane network in practice. We deployed Ethane in our department's 100-Mb/s Ethernet network. We installed 11 wired and 8 wireless Ethane Switches. There were approximately 300 hosts, with an average of 120 hosts active in a 5-min 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, nonservers (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 nonphones and are statically bound to a single access point to prevent mobility (for E911 location compliance). Our policy file is 132 lines long.

VI. 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?*

A. Controller Scalability

Recall that our Ethane prototype is currently used by approximately 300 hosts, with an average of 120 hosts active in a 5-min window. From these hosts, we see 30–40 new flow requests per second (Fig. 5) with a peak of 750 flow requests per second.[14]

[14]Samples were taken every 30 s.

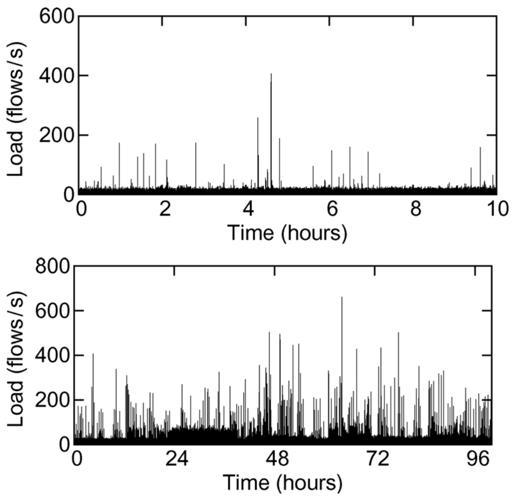

Fig. 5. Frequency of flow-setup requests per second to Controller over a 10-h period (top) and 4-day period (bottom).

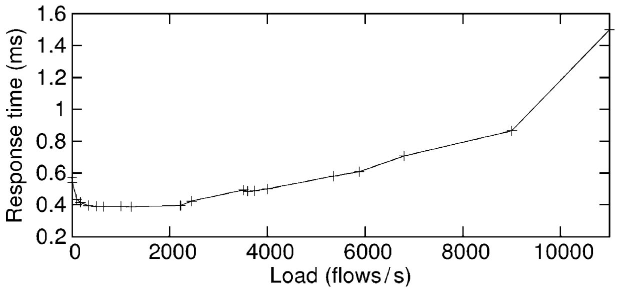

Fig. 6. Flow-setup times as a function of Controller load. Packet sizes were 64, 128, and 256 B, evenly distributed.

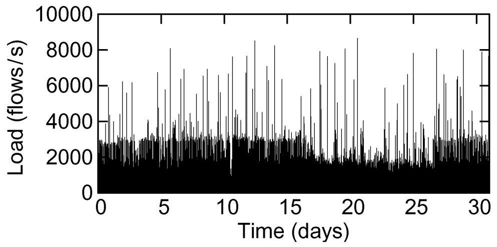

Fig. 7. Flow-request rate for Stanford network.

Fig. 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 ms 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 consider a dataset from a 7000-host network at Stanford, which includes all internal and outgoing flows (not including broadcast). We find that the network during the data collection period has a maximum of under 9000 new flow-requests per second (Fig. 7).

Perhaps surprisingly, our results suggest that a single Controller could comfortably manage a network with over 5000

TABLE I
COMPLETION TIME FOR HTTP GETS OF 275 FILES DURING WHICH THE PRIMARY CONTROLLER FAILS ZERO OR MORE TIMES. RESULTS ARE AVERAGED OVER 5 RUNS

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

hosts without requiring replication. Indeed, flow setup latencies for continued load of up to 6000/s are less than 0.6 ms, equivalent to the average latency of a DNS request within the Stanford network. Flow setup latencies for load under 2000 requests per second are 0.4 ms; 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 Section III-E, one of which we examine next.

B. Performance During Failures

Because our Controller implements cold-standby failure recovery (see Section III-E), a Controller failure will lead to interruption of service for active flows and a delay while they are reestablished. To understand how long it takes to reinstall the flows, we measured the completion time of 275 consecutive HTTP requests, retrieving 63 MB in total. While the requests were ongoing, we crashed the Controller and restarted it multiple times. Table I 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 Section III-E).

Link failures in Ethane require that all active flows recontact the Controller in order to reestablish 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 Fig. 9. In all cases, the path reconverges in under 40 ms, 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.

C. 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. Fig. 8 shows how many active flows we saw in our Ethane deployment; it never exceeded 500. With a table of 8192 entries and a two-function hash-table, we never encountered a collision.

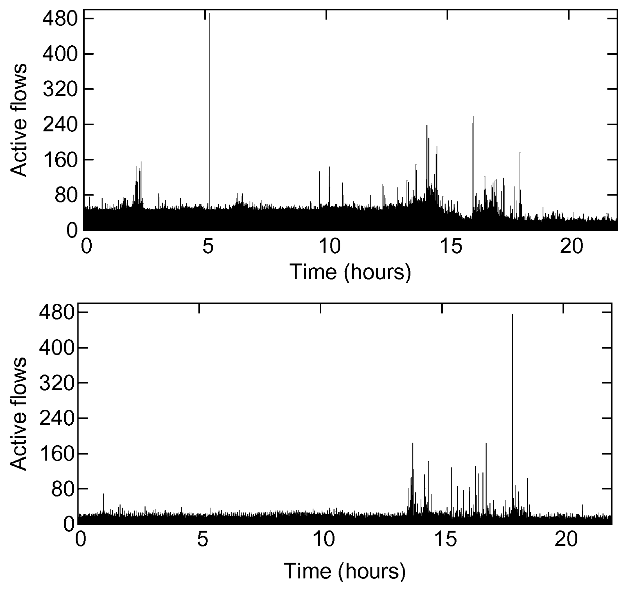

Fig. 8. Active flows through two of our deployed switches.

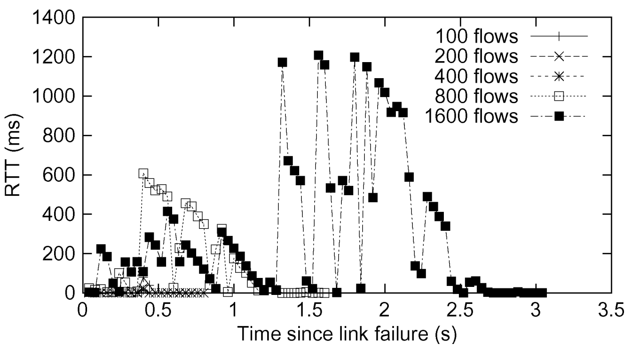

Fig. 9. Round-trip latencies experienced by packets through a diamond topology during link failure.

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). In the University network, our deployed Switches had 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; in our second deployment, the number of ongoing flows averaged 4 per active host.) 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 a flow table capable of holding 8–16 K entries. If we assume that each entry is 64 B, such a table requires about 1 MB of storage, or as much as 4 MB if we use a two-way hashing scheme [17]. A typical commercial enterprise Ethernet switch today holds hundreds of thousands of Ethernet addresses (6 MB, but larger if hashing is used), and millions of IP prefixes (4 MB of TCAM), 1–2 million counters (8 MB 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.

VII. 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 [18], [19]; 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 [10], 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 multihomed 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 nonstandard port numbers. 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.

VIII. RELATED WORK

Ethane embraces the 4D [9] philosophy of simplifying the data plane and centralizing the control plane to enforce network-wide goals [20]. 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, multiservice fast path to traditional IP routers [21]. 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 [22], 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 can be extended to the switches; if one switch fails to enforce a policy, subsequent switches along the path will enforce it. With per-switch enforcement of each flow, Ethane provides maximal defense in depth. Second, 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.

FSL evolved from *Pol-Eth*, the original Ethane policy language [23]. *FSL* extends *Pol-Eth* by supporting dynamic group membership, negation, automated conflict resolution, and distributed authorship. It differs from *Pol-Eth* in that actions only apply to unidirectional flows.

VLANs are widely used in enterprise networks for segmentation, isolation, and enforcement of coarse-grain policies; they are commonly used to quarantine unauthenticated hosts or hosts without health "certificates" [24], [25]. 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., [26]) or secure AAA servers (e.g., [27]). 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.

IX. CONCLUSION

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 and 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 thousands of 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. Furthermore, 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 also anticipate some innovation in Switches. 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 queuing in support of priorities, traffic isolation, and rate control. Our results suggest that even if the Switch does per-flow queuing (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.

Acknowledgment

The authors thank T. Garfinkel, G. Watson, D. Boneh, L. Glendenning, J. Lockwood, and A. Akella for their valuable input. T. Hinrichs and J. Mitchell were heavily involved in the design of *FSL*. D. Mazières, N. Zeldovich, J. Rexford, S. Goldberg, C. Kim, and A. Feldman offered very helpful feedback on early drafts of this paper.

References

[1] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford, "The cutting edge of IP router configuration," *Comput. Commun. Rev.*, vol. 34, no. 1, pp. 21–26, 2004.

[2] T. Roscoe, S. Hand, R. Isaacs, R. Mortier, and P. Jardetzky, "Predicate routing: Enabling controlled networking," *Comput. Commun. Rev.*, vol. 33, no. 1, 2003.

[3] 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*, Sep. 2004, pp. 27–40.

[4] A. Wool, "The use and usability of direction-based filtering in firewalls," *Comput. Security*, vol. 26, no. 6, pp. 459–468, 2004.

[5] A. Wool, "A quantitative study of firewall configuration errors," *Computer*, vol. 37, no. 6, pp. 62–67, 2004.

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

[7] *"Alterpoint,"* [Online]. Available: http://www.alterpoint.com/

[8] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford, "The cutting edge of IP router configuration," *Comput. Commun. Rev.*, vol. 34, no. 1, pp. 21–26, 2004.

[9] 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," *Comput. Commun. Rev.*, vol. 35, no. 5, pp. 41–54, Oct. 2005.

[10] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker, "SANE: A protection architecture for enterprise networks," in *Proc. USENIX Security Symp.*, Aug. 2006, vol. 15, Article No. 10.

[11] T. Hinrichs, N. Gude, M. Casado, J. Mitchell, and S. Shenker, "Practical declarative network management," presented at the ACM Workshop: Res. Enterprise Netw., 2009.

[12] C. Demetrescu and G. Italiano, "A new approach to dynamic all pairs shortest paths," in *Proc. STOC*, 2003, pp. 159–166.

[13] L. Lamport, "The part-time parliament," *Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.

[14] *"NetFPGA,"* [Online]. Available: http://NetFPGA.org

[15] *"OpenWRT,"* [Online]. Available: http://openwrt.org/

[16] *"BerkeleyDB,"* [Online]. Available: http://www.oracle.com/database/berkeley-db.html

[17] A. Z. Broder and M. Mitzenmacher, "Using multiple hash functions to improve IP lookups," in *Proc. IEEE INFOCOM*, Apr. 2001, pp. 1454–1463.

[18] R. J. Perlman, "Rbridges: Transparent routing," in *Proc. INFOCOM*, Mar. 2004, pp. 1211–1218.

[19] A. Myers, E. Ng, and H. Zhang, "Rethinking the service model: Scaling Ethernet to a million nodes," presented at the HotNets, Nov. 2004.

[20] 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," presented at the HotNets, Nov. 2004.

[21] P. Newman, T. L. Lyon, and G. Minshall, "Flow labelled IP: A connectionless approach to ATM," in *Proc. INFOCOM*, 1996, vol. 3, pp. 1251–1260.

[22] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith, "Implementing a distributed firewall," in *Proc. ACM Conf. Comput. Commun. Security*, 2000, pp. 190–199.

[23] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," in *Proc. SIGCOMM*, Kyoto, Japan, Aug. 2007, pp. 1–12.

[24] *"Microsoft Network Access Protection,"* [Online]. Available: http://www.microsoft.com/technet/network/nap/default.mspx
[25] *"Cisco Network Admission Control,"* [Online]. Available: http://www.cisco.com/
[26] *"Consentry,"* [Online]. Available: http://www.consentry.com/
[27] *"Identity Engines,"* [Online]. Available: http://www.idengines.com/

Martín Casado (S'99) received the Ph.D. degree from Stanford University, Stanford, CA, in 2007.

At Stanford, he was a DHS Fellow and was awarded the IBM research fellowship. Prior to attending Stanford, he held a research position at Lawrence Livermore National Laboratory, Livermore, CA, where he worked on information assurance and network security. In 2006, he co-founded Illuminics Systems, an IP analytics company that is now part of Quova Inc. In 2007, he co-founded Nicira Networks Inc., where he currently works as CTO.

Michael J. Freedman (S'99–M'09) received the B.S. and M.Eng. degrees from the Electrical Engineering and Computer Science Department of Massachusetts Institute of Technology, Cambridge, and the Ph.D. degree in computer science from New York University's Courant Institute, during which he spent two years at Stanford University, Stanford, CA.

He is an Assistant Professor of Computer Science at Princeton University, Princeton, NJ. He has developed and operated several self-managing systems—including CoralCDN, a decentralized content distribution network, and OASIS, an open anycast service—that serve more than a million users daily. In 2006, he co-founded Illuminics Systems, an IP analytics company which is now part of Quova Inc. His research focuses on many aspects of distributed systems, security, and networking.

Justin Pettit received the M.S. degree in computer science from Stanford University, Stanford, CA, in 2009.

Before returning to academics to pursue the Master's degree, he spent nearly 10 years in industry working primarily on computer security issues. During that time, he co-founded Psionic Software (acquired by Cisco) and worked at WheelGroup (acquired by Cisco) and IntruVert Networks (acquired by McAfee). In 2007, he joined Nicira Networks as a founding employee.

Jianying Luo received the B.S. degree from Peking University, Beijing, China, in 1996, and the M.S. degree from University of Texas at Austin in 1997, both in computer science.

He is currently a Ph.D. candidate in electrical engineering at Stanford University, Stanford, CA. He has worked on the development of the Cisco Catalyst 4000/4500 series switch as ASIC Engineer since 1999. His research interests include packet-switching architectures and network algorithms.

Natasha Gude received the B.S. degree in computer science from Stanford University, Stanford, CA, in 2007.

Her research while at Stanford focused on building runtime policy engines for enterprise networks. Her work has been integrated into a number of commercial systems. She is a founding employee of Nicira Networks and continues to pursue her Master's degree at Stanford.

Nick McKeown (S'91–M'95–F'05) received the B.E. degree from the University of Leeds, Leeds, U.K., in 1986, and the M.S. and Ph.D. degrees from the University of California, Berkeley, in 1992 and 1995, respectively.

He is a Professor of Electrical Engineering and Computer Science and Faculty Director of the Clean Slate Program at Stanford University, Stanford, CA. From 1986 to 1989, he worked for Hewlett-Packard Labs in Bristol, England. In 1995, he helped architect Cisco's GSR 12000 router. In 1997, he co-founded Abrizio Inc. (acquired by PMC-Sierra), where he was CTO. He was co-founder and CEO of Nemo ("Network Memory"), which is now part of Cisco. His research interests include the architecture of the future Internet and tools and platforms for networking teaching and research.

Prof. McKeown is the STMicroelectronics Faculty Scholar, the Robert Noyce Faculty Fellow, a Fellow of the Powell Foundation and the Alfred P. Sloan Foundation, and recipient of a CAREER award from the National Science Foundation. He is a Fellow of the Royal Academy of Engineering (U.K.) and the Association for Computing Machinery. In 2000, he received the IEEE Rice Award for the best paper in communications theory. In 2005, he was awarded the British Computer Society Lovelace Medal, and the IEEE Kobayashi Computer and Communications Award in 2009.

Scott Shenker (F'00) received the Sc.B. degree from Brown University, Providence, RI, and the Ph. D. degree from the University of Chicago, Chicago, IL.

He. spent his academic youth studying theoretical physics, but soon gave up chaos theory for computer science. His research over the years has wandered from Internet architecture and computer performance modeling to game theory and economics. He currently splits his time between the Computer Science Department and International Computer Science Institute (ICSI) at the University of California, Berkeley.