Scalable Network Management with Merlin

Robert Soulé Shrutarshi Basu Emin Gün Sirer Nate Foster
Cornell University
{soule,basus,egs,jnfoster}@cs.cornell.edu

Abstract

This paper presents the Merlin network management framework. With Merlin, network administrators express functionality such as accounting, bandwidth provisioning, and traffic filtering in a high-level policy language, and use automated tools and mechanisms to implement them. The framework includes: (i) a declarative language for specifying policies, (ii) infrastructure for distributing, refining, and coordinating enforcement of policies, and (iii) a run-time monitor that inspects incoming and outgoing traffic on end hosts. We describe Merlin's policy language and enforcement infrastructure, illustrate the use of Merlin on case studies, and present experimental results demonstrating that Merlin is more efficient and scalable than equivalent implementations based on programmable switches and centralized middleboxes. Overall, Merlin simplifies the task of network administration by providing high-level abstractions and tools for specifying and enforcing rich network policies.

1. Introduction

Programmable network platforms, such as those based on the OpenFlow protocol [17], have made great strides towards simplifying network administration. These platforms provide APIs that let administrators manage forwarding rules, gather traffic statistics, and offer basic quality-of-service guarantees. The key advantage of these platforms is that they provide a single point of control that saves administrators from having to manually manage and configure each individual device in the network. Their success is a testament to the difficulties of managing large-scale networks, as well as the need for automated tools and mechanisms to ease the burden for network administrators.

Expanding on these basic building blocks, this paper aims to further raise the level of abstraction for programmable networks and enable the rich functionality required in networks today. As motivation, consider the functionality that data center networks such as Amazon's EC2 [6] must provide:

- *Accounting:* To keep track of usage for billing purposes, network administrators must record traffic statistics that distinguish the in-network and out-of-network traffic for each customer.
- *Traffic filtering:* To protect end hosts, network administrators must install filtering rules on routers and switches that detect and block malicious traffic.
- *Resource provisioning:* To ensure that important resources such as bandwidth are allocated in accordance with service-level agreements (SLAs), network operators must configure port queues and traffic shapers on devices throughout the network.

Implementing each of these pieces of functionality on their own is not impossible, but when taken together, they become quite challenging. Moreover, this list is by no means exhaustive; most networks pose unique requirements that necessitate additional packet-processing functionality.

In current networks, administrators must typically implement rich functionality by forwarding traffic through specialized middleboxes. Whether they configure the individual network devices by hand, or write a controller program, there is a disconnect between the high-level functionality that operators want enforced (e.g., all packets should be filtered against a given pattern) and how that functionality is realized (e.g., by configuring switches and routers so that all packets traverse at least one middlebox). Moreover, the middleboxes and the controller program have the potential to become bottlenecks as the network scales. Even worse, to actually establish that the desired functionality is implemented correctly, the operator must reason about multiple pieces of software, including the controller program, the switch and middlebox configurations, and the interactions between them via asynchronous network events. Although this approach can be made to work, it is difficult, time consuming, and error prone.

This paper presents the Merlin network management framework. With Merlin, network operators specify the intended behavior of the network using a high-level policy language. These policies are automatically implemented by mechanisms and tools provided by the system. The Merlin system has three components: (i) a *declarative language* for specifying policies and re-

source constraints, (ii) a collection of tools for distributing, refining, and coordinating enforcement of policies called *negotiators*, and (iii) a *distributed enforcement* mechanism that ensures compliance with desired policies while providing scalable performance.

The Merlin policy language provides a suite of expressive constructs for classifying and processing packets. These constructs can be used to describe a wide range of network policies. Policies may contain both mandatory and discretionary components. Merlin policies are ultimately evaluated by a small interpreter installed either in the kernel of end hosts (or on first-hop switches) to determine which packets should be allowed to traverse the network. To ensure that Merlin policies are safe to execute in the kernel, the Merlin language is carefully designed to eliminate undesirable behaviors. For example, it is impossible to write programs with infinite loops, buffer overruns, or null dereferences.

Policy transformations in Merlin are implemented using the infrastructure provided by negotiators. These transformations accomplish three goals: First, they allow tenants to modify discretionary policies to suit their own behavior. Second, they apply explicit constraints that mediate access to shared resources. Third, they transform network-wide policies into local sub-policies that can be enforced on individual nodes in the network. This allows Merlin to use a distributed enforcement mechanism that eliminates bottlenecks and provides scalable performance.

Merlin enforcers guarantee that network traffic complies with the overall policies set down by network administrators. Each enforcer is an interpreter for Merlin policies. Enforcers are interposed on all packets entering and exiting a host to determine if they should be allowed. The policy transformations implemented by negotiators ensure that enforcers do not need access to global state to check compliance with the overall policy. In principle, Merlin enforcers could be deployed in a variety of ways: on end hosts, using middleboxes, or using a combination of the two. However, in this paper, we focus on fully-distributed architectures in which enforcers execute directly on end-hosts.

We have used the Merlin framework to implement a range of practical policies that demonstrate the expressiveness of the system. Our performance evaluation demonstrates that Merlin imposes low overheads on end-hosts and, for sufficiently rich policies, scales significantly better than equivalent middlebox or OpenFlow-based implementations. For heavily loaded networks, Merlin can reduce the latency of transmitting packets by up to 95%. Overall, this paper makes the following contributions:

- It identifies the requirements for a declarative language to manage networks, and describes a practical realization of a language that meets those requirements.
- It presents techniques for distributing policies, delegating policies, and mediating access to shared resources, and an implementation of these techniques.
- It describes a fully-distributed enforcement mechanism, and experimental evidence demonstrating that it scales significantly better than centralized implementations using middleboxes and programmable switches.

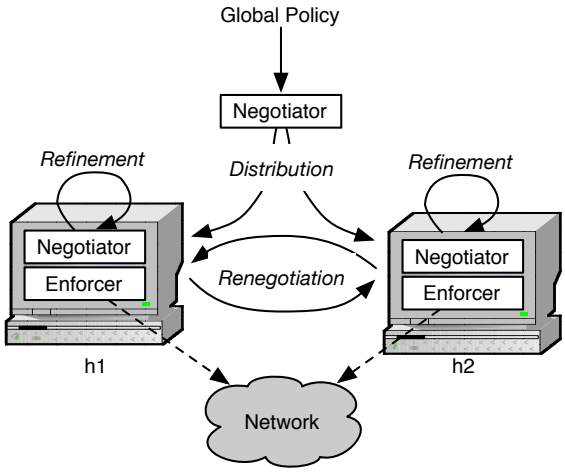


Figure 1: Merlin provides operators with a single point of control for expressing policies. The negotiator tranforms policies so that they can be distributed and refined. Distributed enforcement provides a scalable alternative to centralized mechanisms.

The next section (§ 2) discusses the requirements that led to the overall design for Merlin. The following sections discuss the three main components of Merlin in detail: the language for specifying network policies (§ 3); negotiators that distribute and refine policies, coordinating access to shared resources (§ 4); and enforcers that interpose on all traffic to ensure compliance with the specified policy (§ 5).

2. System Design

The overall goal of Merlin is to create a system that allows operators to express policies in a declarative manner, and to provide tools and mechanism that automatically realize those policies. Any system designed for this purpose must meet the following requirements:

- *Expressive:* Operators need high-level abstractions that allow them to specify a wide range of global network policies clearly and concisely.

Motivation	Requirement	Merlin Feature
Administrators should be able to declare a wide range of policies.	*Expressive*	Declarative policy language with operators for classifying and constraining streams of packets.
Administrators should write one program to configure the network	*Single point of control*	Language support for global policies; infrastructure automatically distributes and enforces policies.
Provide good performance as traffic load grows.	*Scalable*	Run-time enforcer components are fully distributed on end hosts.
Support federated networks with multiple administrative domains.	*Extensible*	Negotiator component supports policy refinement, and reconciling demands for network resources.
Must not leak any information about the network.	*Privacy preserving*	System guards against what distributed mechanisms can reveal.
Must provide guarantees that policies are enforced.	*Verifiable*	Policy transformations can be automatically verified with a theorem prover.

Table 1: Requirements and how they impact the Merlin design.

- *Single point of control:* Operators need to be able to specify the behavior of the entire network without having to separately configure each individual device.
- *Scalable:* To provide good performance, the implementation of the system must scale as the traffic load grows.
- *Extensible:* Because networks typically involve multiple tenants, the system must support both *mandatory* policies, which are specified by an administrator and cannot be overridden, as well as *discretionary* policies, which can be modified by tenants.
- *Privacy preserving:* The system must not expose facts about the network such as its topology or network-wide traffic statistics that would not otherwise be visible to tenants.
- *Verifiable:* The system must provide a basis for trusting that network-wide policies are enforced correctly.

Table 1 summarizes how these requirements impact the overall design of the Merlin architecture, which is illustrated in Figure 1.

Merlin provides a domain-specific language for specifying global network policies. The language allows operators to express useful constraints on sets of packets over time (for example, bandwidth caps). More specifically, operators can classify network traffic into sets using constructs that distinguish packets by location, windows of time, and predicates on header fields and protocols. These sets are then reduced to scalar values using built-in aggregate functions, and constrained using logical operators. Overall, Merlin policies provide an intuitive way to specify rich predicates over streams of network traffic.

Merlin's policy language is built to support two key design choices. First, the use of network locations and high-level constraints ensures that policies are amenable to distribution, refinement, and coordination. Although the language can express stateful predicates (e.g., the amount of web traffic per hour must not exceed 5GB), it is purely declarative and does not provide constructs for explicitly manipulating state. Second, the limited interface of constraints over sets of packets allows us to develop enforcement mechanisms that execute on end hosts, without adversely affecting the host itself. For instance, evaluating a policy on a packet always terminates and has predictable memory usage.

The first design choice is realized in a component called a *negotiator*. A negotiator binds the aggregations over packet sets in Merlin policies to explicit, internal state variables. It then translates the constraints expressed in the policy to equivalent constraints on those variables. These variables and constraints enable a negotiator to apply transformations on policies, which serves three purposes: (i) they refine broad administrator-defined policies with specific tenant-specified functionality; (ii) they translate a global policy into a locally enforceable policy for distribution; and (iii) they reconcile demands for shared resources that span multiple hosts.

In complex networks, multiple negotiators may be distributed and arranged into a tree. For example, in the network depicted in Figure 1, there are three negotiators. The top negotiator distributes the global policy to its children. The two sub-negotiators coordinate access to shared resources—the state variables in a policy, which make resource constraints explicit. To assign these resources, negotiators may impose constraints on the state variables, and then solve them to find sound assignments. The assignment must be compliant with the global policy, but need not be optimal. Hence, negotiators may renegotiate a different assignment amongst themselves without breaking compliance.

The second design choice is realized in the form of Merlin *enforcers*—interpreters for transformed policies. Enforcers interpose on all network traffic, and evaluate each packet with respect to their policies. The evaluation determines whether or not the packet can be forwarded. Because of the distribution transformation, enforcers access only local state during an evaluation. Collectively, the enforcers implement a distributed runtime monitor that is decentralized and scalable.

3. Policy Language

The Merlin policy language is inspired by previous work on streaming query languages [1, 4] and is designed to strike a balance between expressiveness and simplicity. It provides a collection of declarative constructs for describing network policies including operators for classifying streams of network traffic into sets of packets, for aggregating those sets to scalar values, and for expressing logical constraints over those values. At the same time, the language is designed to be simple, in order to make policies amenable to analysis and transformation by negotiators, and to ensure that they can be safely interpreted on hosts.

As an example to illustrate the main features of the language, suppose we want to place a bandwidth cap of 5GB/min for all incoming HTTP traffic to a pair of hosts `h1` and `h2`. In Merlin, we can express this cap using the following policy:

```
(bytes
   { location { h1, h2 } and
    eth.type ~ 0x0800 and
    ip.proto ~ 0x06 and
    tcp.src ~ 80 over
    tumbling(60) }) < 5GB
```

The policy is made up of several nested constructs. The inner-most construct describes a set of packets: all packets on hosts `h1` and `h2` with TCP source port 80 sent or received in the last 60 second tumbling window. Next, the aggregate function `bytes` reduces this set to a scalar value by taking the sum of the sizes of all packets in the set. Finally, the logical constraint stipulates that this scalar must be less than 5GB. Although simple, this policy already describes a non-trivial constraint involving the recent traffic across multiple hosts.

3.1 Syntax

Figure 2 presents the abstract syntax of a core fragment of Merlin's policy language.

Sets. Merlin groups network traffic into sets of packets using combination of predicates on packet headers and payloads, locations, and time. If these components are left unspecified, they match traffic with arbitrary headers and contents, on all hosts, and throughout history.

Numbers	$n ::= 0 \mid 1 \mid 2 \mid \ldots$	
Units	$u ::= \mathsf{B} \mid \mathsf{KB} \mid \mathsf{GB} \mid \ldots$	
Protocol	$r ::= \mathsf{ether} \mid \mathsf{ip} \mid \mathsf{tcp} \mid \ldots$	
Fields	$f ::= \mathsf{src} \mid \mathsf{dst} \mid \ldots$	
Time	$t ::= \mathsf{sliding}(n)$	Sliding
	$\mid \mathsf{tumbling}(n)$	Tumbling
Locations	$l ::= \{\mathsf{host}\}$	Location
	$\mid l \cup l$	Union
Predicates	$p ::= r.f \sim n$	Match
	$\mid \mathsf{location}\ l$	Location
	$\mid p_1\ \mathsf{and}\ p_2$	Conjunction
	$\mid p_1\ \mathsf{or}\ p_2$	Disjunction
	$\mid !\, p_1$	Negation
Packet Set	$s ::= \{p, t\}$	
Term	$t ::= n\ u$	Bytes
	$\mid n$	Literal
Aggregates	$a ::= \mathsf{count}(s)$	Total count
	$\mid \mathsf{bytes}(s)$	Total size
	$\mid \mathsf{average}(s)$	Average size
Expression	$e ::= a$	Aggregate
	$\mid t$	Term
	$\mid e_1 + e_2$	Sum
	$\mid e_1 - e_2$	Difference
Policies	$P ::= e_1 \leq e_2$	Comparison
	$\mid P_1\ \mathsf{and}\ P_2$	Conjunction
	$\mid P_1\ \mathsf{or}\ P_2$	Disjunction
	$\mid !\, P_1$	Negation
	$\mid id\ P_1$	Tagged

Figure 2: Merlin policy syntax.

Predicates. Predicates are built up using standard logical operators such as conjunction (`and`), disjunction (`or`), and negation (`!`). For example, the policy above uses a conjunction of predicates to match all the way up the stack of nested packets: Ethernet, IP, and TCP.

Locations. Merlin supports conventional syntax for Ethernet and IP addresses. Locations are specified using literal addresses, variables, or sets of addresses and variables. Sets of locations can be combined with a union operator. In this paper, we will work with explicit host identifiers. However, the language is designed to work with symbolic sets of hosts given by variables as well, provided the interpretation for those variables is also given. In the example above, the policy is restricted to hosts {`h1, h2`}.

Time. Merlin supports two kinds of time-based windows. Windows declared as `sliding(n)` contain packets from the preceding n seconds. Windows declared as `tumbling(n)` contain packets between the current time and the last multiple of n. Intuitively, a sliding

window advances continuously, while a tumbling window "falls over" every n seconds. In all cases, windows describe a function from potentially infinite streams of packets to finite lists of packets. The example above uses a tumbling window of 60 seconds.

Fields. A predicate of the form $r.f \sim n$ describes the set of packets where field f of protocol layer r matches n. The field f can either be a standard field in a protocol layer, or the actual payload of the packet. The third line of the example matches packets where the Ethernet type is set to IP (i.e., `0x800`). Merlin provides definitions for standard protocols including Ethernet, IP, TCP, and UDP.

Aggregates. To express constraints over sets, a policy must first reduce the set to a scalar value using an aggregate function. Merlin supports several aggregate functions including `count`, which computes the number of packets in the list, as well as `bytes` and `average`, which compute the total and average size of packets in the set respectively.

Constraints. At the top-level, a Merlin policy comprises a logical constraint over linear combinations of aggregate values. In this way, policies can specify global predicates that must hold about the stream of traffic in the network. In the example above, the constraint limits the total size of all HTTP traffic to 5GB.

Tags A tag is an optional identifier indicating that a policy is discretionary. Semantically, a tag declares that a policy constraint is unspecified, and that a tenant is free to apply any constraints they wish. Operationally, a tag provides a way to reference a policy that can be modified, like a capability.

3.2 Policy Language Discussion

The Merlin language emphasizes direct expression of network policy and simplicity. The language provides programmers with a collection of constructs for directly expressing the intended behavior of the network (as described above). These constructs are powerful enough to allow Merlin policies to express a wide range of network functionality (as demonstrated in Section 7.1).

The language is, by design, *not* a general purpose programming language. To enforce policies, Merlin programs are evaluated by an interpreter that runs inside the kernel of the host machine (described in depth in Section 5). Therefore, safety is a crucial design consideration, and Merlin intentionally provides a limited programming interface. Notably, Merlin lacks an assignment operator, and does not provide any programmer-accessible mutable state. The language presents an abstraction that each policy maintains its own copy of the packet set, although internally sets may share pointers. As a result, executing Merlin policies does not produce side-effects. Moreover, the language omits several features that are a common source of errors: there are no loops that might not terminate; no arrays that might be accessed by an out-of-bounds index; and no pointers that might reference null addresses.

The Merlin language raises the level of abstraction provided to operators while enabling powerful underlying mechanisms to enforce network policy in novel ways. We explore these mechanisms further in the following sections.

4. Negotiators

Merlin negotiators implement transformations on policies. They map policies to sets of resource constraints, and allocate global resources to network tenants. This allows negotiators to transform global policies into a form that can be independently enforced on each end-host with minimal coordination. Negotiators are hierarchical, so that parents impose constraints on their children. Children can refine their own policies, as long as the refinement implies the parent policy. Peers are free to renegotiate resource allocations, unless the new allocation would violate parent constraints.

4.1 Fine-Grained Resource Allocation

Before transforming a policy, the negotiator must first bind the aggregated packet sets in the Merlin language to explicit *state variables*. These state variables represent a particular allocation of network resources. Each aggregated set is assigned a fresh variable, and the variables are substituted into the contraints specified by the policy.

For the bandwidth cap example in Section 3, the negotiator might produce a variable `x`, and a constraint:

$$x < 5GB$$

In this case, the variable `x` represents the allocation of resources to the coarse set of hosts `h1` and `h2`. The equation states that the allocated bandwidth usage must be less than $5GB/min$.

Specialization To distribute policies to end-hosts, a negotiator *specializes* a policy. To specialize a policy, the negotiator produces additional state variables corresponding to finer-grained allocations for each particular host. It then emits a revised linear constraint which states that the sum of the variables must be less than the original bound.

For example, to distribute the bandwidth cap policy from the previous section, the negotiator might generate new state variables, `y` and `z`, and a new formula:

$$y + z < 5GB$$

where y corresponds to the bandwidth allocated to `h1`, z corresponds to the bandwidth allocated to `h2`, and the sum of the allocations must be less than $5GB$ at all times.

It is easy to see that by enforcing the specialized policy on each host, the global policy will also be correctly enforced.

Constraint management. The key challenge for distribution is to find an assignment of values to state variables that accomplish the policy goals. To do this, the negotiator can solve the constraints to find any satisfying assignment. As a default heuristic, Merlin attempts to find solutions with equal values by minimizing the Euclidean distance between the variable assignments. For the example, the solver produces a solution where both y and z are bound by 2.5GB. The result is a set of three constraints:

$$y + z < 5GB$$
$$z < 2.5GB$$
$$y < 2.5GB$$

But in fact any solution will do—any formula that implies the original formula will guarantee that the original global policy is correctly enforced. Note that it is easy to express particular allocations as constraints within the Merlin language itself. This allows the negotiator to assign allocations such as "one host gets twice the bandwidth of another," for example.

Distribution. The last step in the transformation is to re-write the policies in terms of the new variables and assignments. In other words, given a single policy and a set of n hosts, the transformation will produce n policies, one for each host, that collectively imply the original. The negotiator then installs the appropriate policy on each host. For example, host `h1` will get a policy that limits the total number of bytes sent on TCP port 80 to be less than 2.5GB in each 60 second interval. Internally, the negotiator keeps track of which hosts have been allocated what resources by mapping hosts to state variables, and state variables to allocations.

4.2 Refinement and Renegotiation.

Merlin is designed to reflect the federated nature of networks today. To that end, Merlin policies can be refined to suit the needs of individual network tenants. Moreover, resource allocations can be renegotiated at the discretion of tenants, so long as they abide by overall network policy. We have implemented several such cooperative renegotiators to share network resources among tenants.

Refinement Network operators often want to delegate policy decisions to network tenants. For example, in Amazon EC2, network instances have certain ports disabled by default, such as port 22 for SSH traffic, but customers are free to modify the default settings to enable those ports.

Merlin uses tagged policies to delegate policy decisions (i.e., to mark them as discretionary). A tagged policy indicates that the global policy is *unspecified* for certain set of packets. Network tenants may apply any constraints they wish to a tagged policy.

In general, negotiators can modify any policy, tagged or not, if the new policy implies the old one. That is, a policy can always be made more restrictive.

Renegotiation Merlin negotiators allocate resources based on the constraints in a policy. However, there are many situations when hosts may wish to change their allocations. For example, due to transient load, a host might need to send more data than its current constraint allows. Merlin allowing tenants to change their allocations through a renegotiation process. Negotiators exchange messages amongst themselves to agree on a new allocation. Once an agreement has been reached, the allocations can be adjusted in two ways: through a central negotiator that acts as a broker, or in a peer-to-peer fashion between participant negotiators. Figures 3a and 3b illustrate these two approaches.

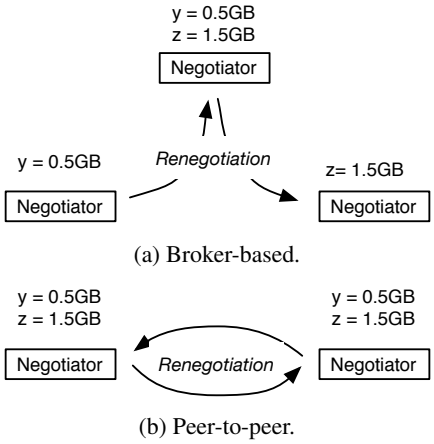
In either case, the new allocation must conform to the parent policy. A parent policy may apply to many hosts, not just the participants in the renegotiation. At a minimum, renegotiation requires the *form* of the parent constraint, the allocations of the participants, and mappings of variables to participant hosts. Therefore, the knowledge revealed during negotiation is restricted to information about the participant hosts, and the global policy.

Because some knowledge is necessary for renegotiation, choosing between the broker and peer-to-peer strategy involves a tradeoff between performance and privacy. The central broker approach is privacy-preserving, but adds the overhead of working through the broker. The peer-to-peer approach avoids the broker overhead at the expense of revealing information to end-hosts.

Merlin does not specify a protocol for hosts to reach an agreement on a new allocation. The details of such a protocol are dependent on a variety of factors, such as the trust relationship between tenants, and the tolerance for time spent reaching a consensus. These are exogenous concerns better handled outside of the core system.

Merlin does provide the mechanism and means for verifying a new allocation. In our evaluation, we use a centralized negotiator, and a simple renegotiation protocol that assumes cooperative peers request reallocations in the collective best interest.

One of the key features of Merlin is that negotiators do not need to be trusted. All of the negotiators just described have the property that they only change the granularity and restrictiveness of the policies they receive as input. Hence, if the conjunction of the formula contained in the output policies implies the formula in the input, then the correctness of the global policy is guaranteed. Moreover, because formulas are simple linear constraints over variables, we can decide implications using an automatic theorem prover. In our implementation, we use



(a) Broker-based.

(b) Peer-to-peer.

Figure 3: Broker-based and peer-to-peer re-negotiation.

the Z3 SMT solver to decide implications between formulas [18].

4.3 Negotiator Discussion

Merlin negotiators transform the high-level operator-specified policies to achieve three goals. First, they specialize policies for distributed enforcement. Second, they refine policies for delegation. Third, they renegotiate policies to modify resource allocations. All three transformations are achieved with the same technique: change the granularity of the sets in the policy, apply appropriate constraints, and reassign resources. This technique provides a powerful and verifiable base layer for policy management. The resulting system gives as much freedom as possible to network tenants, while still ensuring that global policies are respected.

5. Enforcers

Merlin enforcers are interpreters for policies, charged with ensuring policy compliance. They evaluate each network packet with respect to the specified policies, and determine if it should be forwarded. Although Merlin policies typically involve network-wide properties encompassing large numbers of hosts, each individual enforcer does not see and is not responsible for large-scale policy enforcement. Instead, the Merlin negotiator sends to the enforcer only those policies that concern the packets destined to or emanating from a specific host in its purview.

End-host enforcers are written as loadable kernel modules. Our Linux-based implementation uses the `netfilter` callback functions for access to packets on the network stack. The interface to the enforcer is small: it takes as input a packet and returns a boolean value indicating if the the host should drop the packet or allow it to proceed.

The enforcer is designed to have minimal dependencies on operating system services in order to make it portable across different operating systems. Our implementation of the Merlin enforcer requires only about a dozen function calls to be exported from the operating system to the enforcer, relating to bookkeeping issues such as memory allocation and synchronization.

5.1 Expressiveness

Merlin can enforce network properties that are expressible as functions on histories of packets. This means that Merlin enforcers are limited by the information available in the packet contents. Without any changes to the network, Merlin can provide a broad range of typical functionality, including statistics gathering, traffic filtering, and provisioning.

Merlin can enforce additional functionality if more information is available in the packets. For example, if the network employs a source-routing scheme, such as pathlet routing [10] or a forwarding fabric [3], Merlin can use the information embedded in tags to enforce routing or path properties. As a proof-of-concept, we have implemented a simple version of source-routing in which paths through the network are enumerated and stored in an MPLS label. With this scheme in place, we have used Merlin to enforce traffic isolation properties.

5.2 State Management

The biggest challenge for the design of the enforcer is managing the state required to enforce properties. Merlin uses standard mechanisms, such as a routine to clean up stale windows periodically, and an optional runtime memory limit that will cause Merlin to drop packets until windows expire.

Beyond these mechanisms, Merlin includes optimizations for handling sliding windows. Unlike aggregations over tumbling windows, which can be computed incrementally, aggregations over sliding windows require maintaining histories of packets. This is a well-known problem in streaming systems [13]. A naïve implementation of sliding windows would store packets proportional to window sizes.

To guard against excessive memory usage, Merlin sliding windows include a parameter for *summarizing* packets. Packet summaries enable sliding windows to advance by the amount of time specified by the summary variable, instead of continuously with time. Packets within a summary are aggregated together to reduce memory usage. Put another way, a summary turns a sliding window into a sequence of tumbling windows.

Furthermore, Merlin treats windows of infinite size as a special case. Because no packets are ever evicted from an infinite window, aggregate information can be

updated incrementally, rather than being computed after each window trigger.

5.3 Trust

Merlin assumes a trusted deployment in which all host machines can be assumed to execute and comply with enforcers. An interesting, but orthogonal, problem is to deploy Merlin in an untrusted environment. Several techniques have been proposed to verify that an untrusted machine is running specific software. Notable examples include proof carrying code [19], and TPM based attestations, such as used by ETTM [5] and the Nexus operating system [23].

To explore the feasibility of an untrusted deployment, we have implemented a version of Merlin that runs on the Nexus operating system. Nexus provides proof in the form of logical attestations that the enforcer is installed and interposes on all network traffic.

Porting the enforcer to Nexus required modifying only 8 lines of code in the kernel, and redefining less than a dozen macros in the enforcer. The modest number of required changes corroborates the assertion that the Merlin enforcer is self-contained, and portable across operating systems.

5.4 Host Failure

Because Merlin negotiators specifically transform policies to avoid centralized coordination, a node failure does not affect overall policy enforcement. In the event of a failure, a particular resource allocation will be underutilized by the group including that node. However, the overall policy will continue to be correctly enforced. Merlin can detect a host failure during the renegotiation process. If a negotiator detects a failure, it can reallocate the resources formerly allocated to that host.

5.5 Enforcer Discussion

Merlin enforcers implement a distributed network runtime monitor. Each individual enforcer is an interpreter for Merlin policylets, and evaluates packets for compliance with global network policy. Enforcers have a small interface, with minimal dependencies on the host operating system. By providing a consistent, yet powerful target platform for the Merlin language, the enforcers enable network administrators to specify their policies at a high level without having to reason about how those policies are enforced. Such a mechanism raises the level of abstraction at which networks may be programmed and opens up the possibility of further transformation, verification, and optimization of the locally-enforced policies.

6. Implementation

We have implemented the Merlin framework described in this paper. The core negotiator functionality required 2,262 source lines of OCaml code. This involves logic to parse and transform policies into policylets, several negotiators, and serialize and deserialize formulas for Z3. An additional 77 lines of code wrapped the core negotiator component and provided network connectivity. Our implementation currently uses the Z3 SMT constraint solver [18] for validating resource assignments. However, the negotiator does not depend on any Z3-specific functionality, and could be replaced with a different solver.

The core enforcer functionality is a policy evaluator, which required 3,228 source lines of C code. The evaluator executes the policy and decides whether or not to forward a packet. To pass packets to the evaluator, we implemented a loadable Linux kernel module using the netfilter callback functions. The kernel module required 102 lines of C code.

We have also implemented a version of the Merlin enforcer for the Nexus trusted operating system [23]. The Nexus version of the enforcer is written as an extension to the Nexus kernel, rather than a loadable module. Porting the evaluator component required redefining 11 macros to use Nexus-specific system calls. Modifying the Nexus kernel to pass packets to the evaluator required changing only 8 lines of code.

7. Evaluation

In evaluating Merlin we explore the following questions: (i) What is the overhead for enforcing policies on the end-hosts? (ii) How does Merlin's distributed enforcement strategy compare to a middlebox and a software-defined network implementing the same policy? (iii) How do different negotiation strategies affect network utilization?

7.1 Enforcement Overhead

The overhead for enforcing policies on end-hosts is dependent on both the complexity of the policy, and the traffic being inspected. Nevertheless, we attempt to quantify this overhead on several realistic policies.

For each policy, we measure the time it takes for the enforcer to evaluate 1 million 1024-byte packets. This experiment shows the overhead of executing the enforcer in isolation, without any interaction with the network. Figure 4 presents the results, which show that even with complex policies, Merlin imposes low overheads. Hosts running Merlin enforcers would still be able to saturate both 1 Gigabit and 10 Gigabit links.

Below, we discuss the policies in detail. For each, we present both the Merlin pseudocode, and a short description. Due to space constraints, and to improve clarity of presentation, we present only the core required functionality.

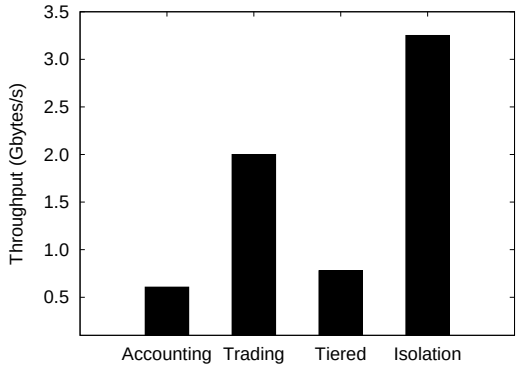


Figure 4: Enforcing Merlin policies adds little overhead.

Accounting. Many networks charge customers based on their usage. Therefore, keeping accurate usage statistics is vital. However, pricing strategies may be more nuanced than simply recording the total usage. For example, a service like Amazon's EC2 charges different rates for sending traffic internally versus externally.

The following example implements an EC2-inspired billing policy. It uses a tumbling window to track the monthly usage. The policy distinguishes traffic by the destination IP address to determine whether or not it falls in the internal subnet (192.168.1.1/16).

```
(print {location 192.168.1.1/16 and
        ip.dst ~ 192.168.1.1/16,
        tumbling(30) } >=0)
and
(print {location 192.168.1.1/16 and
        !(ip.dst ~ 192.168.1.1/16),
        tumbling(30) } >=0)
```

Note that this policy uses an aggregator we have not discussed—`print`, which does does not produce a scalar value but instead prints the total size of the set to a file descriptor. These aggregators are immaterial for policy enforcement but are useful for recording statistics.

Tiered services. Networks commonly provide tiers of service, based on past usage. For example, an ISP such as Time Warner enforces a high bandwidth cap for customers below a certain usage threshold, but enforces a much stricter cap when the threshold is exceeded. The following policy implements a tiered bandwidth cap.

```
(bytes {true, tumbling(600)} <= 20MB =>
 bytes {true, sliding(60) } <= 5MB)
 or
(bytes {true, sliding(60) } <= 1MB)
```

It tracks past usage over a long interval, and applies two different bandwidth caps depending on the result.

Bandwidth trading. Merlin negotiators allow network tenants to renegotiate their resource allocations for the collective good. This is similar to systems such as EyeQ [14], where senders and receivers coordinate individual bandwidth constraints to control congestion.

The following Merlin policy allows an administrator to impose a large bandwidth constraint on a set of hosts, while delegating the ultimate fine-grained resource control to the hosts themselves.

```
bytes {true, sliding(60)} <= 600MB
```

The hosts would then either use a broker or cooperate to subdivide the available bandwidth.

Path isolation. Many regulatory guidelines require corporations to keep different parts of their businesses completely separate. For example, in investment banking, the Sarbanes-Oxley Act requires that the investment side of the company be completely separated from the brokerage side of the company. As a result, corporate networks want to ensure that traffic from different parts of the network never travel along the same path.

Merlin can enforce this type of path isolation property by leveraging prior work on source routing, which encodes the path a packet travels through the network in the packet itself [3, 10]. Our test application assigns each path a separate identifier, and stores the path in an MPLS label. The following policy inspects the label to ensure isolation.

```
(count {location 192.168.1.1/8 and
        ip.dst ~ 192.168.1.1  and
        mpls[0].label ~ 0x88,
        sliding(60)} == 0)
and
(count {location 192.168.1.1/8 and
        ip.dst ~ 192.168.1.2  and
        mpls[0].label ~ 0x77,
        sliding(60)} == 0)
```

The above examples use the constructs described in Section 3 to declaratively specify network behavior. We can directly specify the desired behavior without having to reason about forwarding rules, network state, or the configurations of individual network elements. We can safely delegate enforcement to our in-kernel enforcers with minimal overhead.

7.2 System Comparisons

After evaluating the standalone overhead, we compare Merlin to two current alternatives to providing rich network functionality. First, we consider a middlebox approach where all traffic is routed to a centralized middlebox which implements the required functionality on a

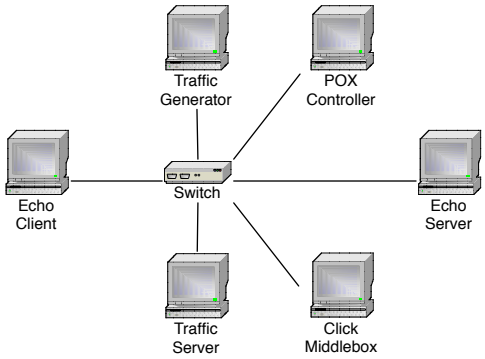


Figure 5: The experimental setup for latency measurements. The traffic client and server generate cross traffic. The latency measurement is taken on the echo client and server. The Click middlebox applies DPI to all packets.

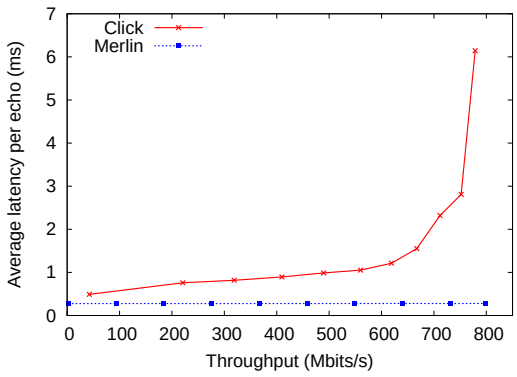


Figure 6: A centralized middlebox implementation of deep packet inspection shows 12 times the latency of an equivalent Merlin implementation.

per-packet basis. Second, we consider an SDN approach where only the *first* packet of each flow is sent to a controller for further processing. The controller installs or removes forwarding rules on a switch depending on the processing results.

7.2.1 Experiment Setup

Figure 5 illustrates the network used in the experiment. It consists of six machines connected by 1-Gigabit links via an OpenFlow-enabled Pronto 3290 switch. Four of the machines generate traffic. A fifth machine acts as a dedicated controller for the switch, and runs the POX [20] controller platform. The sixth machine acts as a dedicated middlebox running the Click modular router [15]. The Merlin measurements did not use the middlebox machine.

To test system scalability, we measured the latency for sending traffic under increasing network load. All traffic was subject to a policy implementing a simple deep packet inspector (DPI). Namely, the policy identified and dropped TCP packets that contain a particular blacklisted URL in their payload.

We collected the latency measurements on a pair of machines exchanging 1000-byte messages. In Figure 5, these machines are labeled Echo Client and Echo Server.

To place load on the network, we used a second pair of machines to generate traffic. In Figure 5, these machines are labeled Traffic Generator and Traffic Server. The generator forked n processes, each continually sending data to the server. We increased the amount of traffic by increasing the number of processes running concurrently. We measured the traffic throughput on these machines, and increased the load until we were unable to induce an increase in throughput. The traffic profile was generated according to the datacenter traffic distribution identified by Greenberg et al. [11].

For each step of increasing load, we took 1000 latency measurements, and computed the 90th percentile. This eliminates extreme outliers due to packet loss and retransmission. We ran each experiment three times, and report the average results.

7.2.2 Middlebox Comparison

We compare the Merlin and middlebox approaches along two axes: system complexity and scalability. It turns out that enforcing even the deliberately simple DPI policy with the middlebox approach increases system complexity and results in significant packet delays when the network is heavily utilized.

System Complexity For the middlebox-based approach, we implemented the DPI policy as a Click router element [15] and configured the network to route all traffic through the middlebox.

The Click element loops over the packet contents searching for the target pattern. Even this simple application required over 60 lines of C++ code, and a separate Click script to connect the element to the network interfaces. Configuring the switch required adding five rules to the controller program to ensure that all traffic passed through the middlebox before being forwarded to the destination.

By contrast, the DPI policy required only one line of code in the Merlin language. The Merlin infrastructure automatically deployed the policy to every end-host. There were also no changes required to the controller program or the network's routing policy.

System Scalability The results are shown in Figure 6. When the network is lightly loaded, the performance of both systems are comparable. This is as expected, since the computation performed by Merlin and the Click ele-

ment are the same. The extra overhead for the middlebox case, about 0.25ms, is due to the extra hops (to and from the middlebox) that each packet needs to travel.

As the network load increases, the effects of the middlebox bottleneck become manifest. When the load is about 750 Mbits/second, the latency spikes to over 6 milliseconds, and we see an increasing number of packet drops and retransmissions.

In contrast, the latency for the Merlin setup stays constant at around 0.26 milliseconds. This is a 95% reduction in latency when the network is heavily loaded. Merlin's distributed enforcers scale far better than the centralized middlebox enforcement strategy.

7.2.3 Software-Defined Network Comparison

To compare the Merlin approach with implementing functionality in an SDN controller, we implemented the same deep packet inspection operation, with one small change. Instead of inspecting every packet sent on the network, the controller only inspects the first packet of every flow. This is more consistent with how controller functionality is deployed in practice.

The switch is configured to send the first packet of each flow to the controller, and the controller searches for the blacklisted pattern. If the pattern is not found, it installs a rule on the switch to directly forward the remainder of the flow from the source to the destination. If the pattern is found, a rule is installed to drop all subsequent packets on the flow.

Controller Implementations Network switches have a limited table size for forwarding rules (typically a few thousand entries). This means that under high load (large numbers of flows), the table size is not sufficient for the number of flows in progress. We implement three separate methods for managing the rule table.

1. *Buffering:* forwarding rules are installed with a timeout. The switch informs the controller when rules timeout. This allows the controller to keep track of how many rules are currently installed. If the rule table is full, the controller delays the installation of new rules until older rules have expired.
2. *Random eviction:* Forwarding rules are installed permanently. The controller keeps track of the number of rules installed. If the rule table is full, it removes a rule at random before installing a new one.
3. *FIFO eviction:* Forwarding rules are installed with a timeout. The controller maintains an ordered list of installed rules and records when each rule is due to timeout. If the rule table is full, it evicts the oldest installed rule that has not already timed out before installing a new rule.

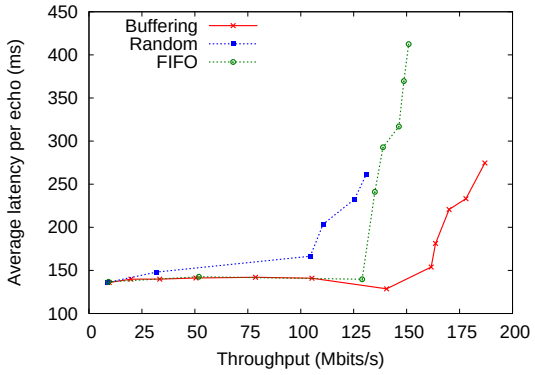


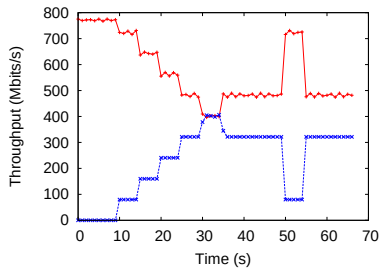
Figure 7: Software-defined network implementations of deep packet inspection shows orders of magnitude times the latency of an equivalent Merlin implementation.

Experimental Results Under light load all three controllers impose about the same latency. However, the latency is over two orders of magnitude higher than either Merlin's deep packet inspection or a middlebox-based equivalent. This is explained by the low bandwidth connection from the switch to the controller (we recorded a maximum of 300 packets a second) and the need to install forwarding rules on the switch before the remainder of the flow can be transmitted.

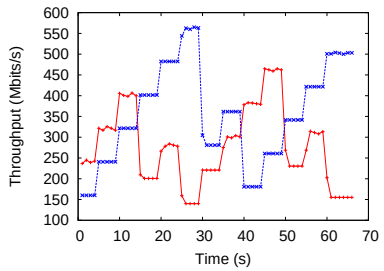
Under high load the three controllers all suffer performance degradation and latency spikes, but with different profiles. The buffering controller provides the greatest throughput before the latency spikes. The timeouts allow the rule table to be cleared without controller intervention, but the controller itself must respond to *both* new flows and timeout notices for expired flows, essentially doubling its load. After the point shown in the graph the controller becomes too heavily loaded to respond reliably to new flows. We observed that latencies become erratic and connections time out.

The random eviction controller provides the least throughput before the latency spikes. This is to be expected, as only a single rule is evicted at a time before installing a new one. Thie limits how fast the controller can respond to increasing numbers of new flows.

The FIFO eviction controller spikes at a throughput in between the other two. Unlike the random eviction controller, it takes advantage of flow expirations and does not always evict rules to free up space in the table. But at the same time, it must manage a complicated data structure and occasionally explicitly evict rules (unlike the buffering controller). The overhead of maintaining this data structure and evicting a rule at a time prevents scaling to higher throughput and causes higher latencies.



(a) Negotiator with a priori knowledge of resource needs.



(b) Negotiator with AIMD adaptation.

Figure 8: Different renegotiation strategies.

While software-defined networks can provide some rich functionality by performing computation at the controller, they incur a high performance penalty (even if processing on a per-flow basis). For all but the most lightly loaded networks, this penalty is unacceptably high. Merlin can implement the same functionality on a *per-packet* basis, and with very low overheads.

7.3 Renegotiation

Merlin negotiators can support a wide range of resource management schemes. We implemented two common approaches: *a priori knowledge of resources* and *additive-increase, multiplicative decrease* (AIMD) adaptation. With a priori knowledge, network tenants declare their resource requirements ahead of time, and the negotiator tries to satisfy their requests. With AIMD, tenants dynamically adjust resource demands, incrementally trying to increasing their allocation.

Figure 8 shows the bandwidth usage over time for two hosts exchanging bandwidth utilization using the two negotiator strategies. Both hosts attempt to send as much UDP traffic as possible to a common server, sharing a 1 Gigabit link. In practice we achieved a maximum throughput of 800 Mbit/s for UDP traffic (even without any enforcers or negotiators). Merlin enforcers impose a bandwidth cap on both hosts, similar to the bandwidth trading example in Section 7.1. The two negotiators adjust their resources using a peer-to-peer negotiator.

1. A priori knowledge. This microbenchmark is shown in Figure 8a. At the beginning of the experiment, host `h1` is the only machine sending traffic, and can send to the maximum shared limit. After 10 seconds, host `h2` joins the network, and begins to ask for increasingly larger bandwidth allocations. This continues until they agree that they have gone too far, and back off, eventually agreeing on a 60/40 split. They stay at this steady-state until `h1` generates a short burst of traffic at 50 seconds, and takes almost of the resources back. Finally, they return to their 60/40 steady state.
2. AIMD adaptation. In Figure 8b, the two hosts dynamically adjust their resources using a congestion control strategy similar to that employed by TCP. At each time step, both hosts try to increase their current allocation by $0.1 * \texttt{limit}$. If that increase would exceed the shared cap, then the host halves its allocation. With this strategy, the hosts alternate maximum allocations over time.

Renegotiation Overhead The time it takes for tenants to renegotiate is dependent on the negotiation strategy they use. Merlin provides the mechanism for setting the new allocations. To measure the overhead for the negotiator mechanism, we sent 1000 negotiation updates from one host to another through a broker negotiator.

The average time to update the new allocations was only 99 μs with a standard deviation of about 19μs. This number is mostly dependent on the time it takes a message to travel from one host to the other through the broker. Updates are performed by writing the new allocation into a file in the `/proc` filesystem exposed by the kernel module.

8. Related Work

Merlin enforcers fall under a class of security mechanisms known as execution monitoring [21]. Tagged policies implement a form of capabilities [16].

Merlin enlists end-hosts to perform distributed enforcement. A number of systems have explored the idea of using end hosts to implement network functionality. One related system to Merlin is End to the Middle (ETTM) [5], a scalable and fault-tolerant network manager that runs software in a trusted execution environment on end hosts. However, while in Merlin the end-host component is a small, trusted interpreter, ETTM allows for arbitrary packet-processing code. This interpreter allows Merlin to define a high-level language for expressing policies, which ETTM does not provide.

Participatory Networking (PANE) [7] addresses a complementary set of management concerns. PANE identifies a standard interface by which an end host

can request functionality from the network. For example, PANE allows a host to explicitly reserve bandwidth along a particular path. However, PANE does not provide a language for specifying network-wide properties, or provide a way to enforce proper use of network resources.

SideCar [22] and Fabric [3] adopt a hybrid network architecture, where the first-hop software switches provide a programmable platform for deploying rich network functionality, but the rest of the network is a simple forwarding fabric. Our implementation of Merlin is inspired by the design of these architectures.

Merlin's policy language is similar in spirit to the high-level languages used in Frenetic [9], Ethane [2], FML [12], and PANE [8].

EyeQ [14] enlists end hosts to help sharing of the network. EyeQ hosts execute a sender module that regulates the rate at which it sends traffic. Sender modules on different hosts provide fine-grained feedback about the set of flows in the network to arbitrate bandwidth and provide congestion control. EyeQ is similar to Merlin in its use of a forwarding fabric, and its use of end hosts to enforce bandwidth properties. Unlike Merlin, EyeQ does not provide a high-level language or compiler, or mechanisms for enforcing other kinds of policies such as filtering and routing.

Active Networks [24] were also based on the idea of using the network as an "interpreter" for "programs" encapsulated in packets. Unlike Merlin, Active Networks envisioned tags as complete programs that could affect both the switch state and the paths packets take through the network. In contrast, Merlin requires essentially no computation on switches.

9. Conclusion

The success of programmable network platforms has demonstrated the benefits of providing high-level abstractions for managing networks. Merlin compliments these approaches by focusing on functionality beyond routing policies. Merlin can be used to gather network statistics, such as for accounting and billing; traffic filtering, like a distributed firewall; or fine-grained resource allocation and coordination. Used in coordination with source-routing techniques, Merlin can enforce path and routing properties.

The high-level design of Merlin, in which trusted interpreters are installed on end-hosts to provide network functionality and enforce policies, is powerful and flexible. The policy transformations provided by Merlin negotiators makes the design feasible. Overall, Merlin's efficient design and concise language greatly simplify the task of network policy enforcement, and lays the foundation for a variety of future research to further explore network programmability.

References

[1] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proc. 21st ACM Symposium on Principles of Database Systems*, pages 1–16, June 2002.

[2] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: taking control of the enterprise. In *Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 1–12, August 2007.

[3] Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *Proc. 2012 ACM Workshop on Hot Topics in Software Defined Networking*, August 2012.

[4] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. ACM SIGMOD*, pages 647–651, June 2003.

[5] Colin Dixon, Hardeep Uppal, Vjekoslav Brajkovic, Dane Brandon, Thomas Anderson, and Arvind Krishnamurthy. ETTM: A scalable fault tolerant network manager. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*, 2011.

[6] Amazon's elastic compute cloud. `http://aws.amazon.com/ec2/`.

[7] Andrew Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shiriam Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. August 2013.

[8] Andrew D. Ferguson, Arjun Guha, Jordan Place, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory networking. In *Proc. 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, 2012.

[9] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: a network programming language. In *Proc. 16th ACM International Conference on Functional Programming*, pages 279–291, September 2011.

[10] P. Brighten Godfrey, Igor Ganichev, Scott Shenker, and Ion Stoica. Pathlet routing. *ACM SIGCOMM Computer Communication Review*, 39(4):111–122, August 2009.

[11] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: a scalable and flexible data center network. In *Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 51–62, 2009.

[12] T. Hinrichs, N. Gude, M. Casado, J. Mitchell, and S. Shenker. Practical declarative network management. In *Proc. 2009 Workshop: Research on Enterprise Networking*, pages 1–10, 2009.

[13] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys*, 2013. To appear.

[14] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazieres, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. EyeQ: Practical network performance isolation at the edge. In *Proc. 10th ACM/USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[15] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[16] Hank Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, Mass, 1984.

[17] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, March 2008.

[18] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[19] George C. Necula. Proof-carrying code. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[20] The pox openflow controller. Available from `http://www.noxrepo.org/pox/about-pox/`.

[21] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[22] Alan Shieh, Srikanth Kandula, and Emin Gün Sirer. Sidecar: building programmable datacenter networks without programmable switches. In *Proc. 9th ACM Workshop on Hot Topics in Networks*, pages 21:1–21:6, 2010.

[23] Emin Gün Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *Proc. 23nd ACM Symposium on Operating Systems Principles*, pages 249–264, 2011.

[24] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of Active Network research. *IEEE Communications Magazine*, 25(1):80–86, January 1997.