in the network. Instead, accepted requests and realized hints determine network policy. We call such accepted requests and realized hints *policy atoms* – units of the overall network policy. Policy atoms are arranged in the same hierarchy as the share tree, forming a *policy tree*. A policy tree is a declarative data structure that represents the desired global policy for the network. PANE materializes this policy in the network by installing rules in the switches that implement an equivalent policy (§6).

Policy atoms thus exist in the context of a share, and are bound by the shares' privileges and flowgroup. However, policy atoms may conflict. For example, one policy atom may deny all HTTP flows, while another allows HTTP flows. These atoms may even exist on different shares. The PANE share tree is flexible: it supports oversubscription, and allows several shares to express policies for overlapping flowgroups. A key novelty of PANE is a principled and intuitive *conflict-resolution* algorithm for hierarchical policies.

We use Hierarchical Flow Tables (HFTs) to materialize PANE's policy tree. HFTs provide a model for resolving conflicts in a hierarchy of policies, and a formally-verified compiler from such hierarchies to flow tables suitable for OpenFlow switches. In particular, HFTs use *conflict resolution operators* within and between each node in the hierarchy to flexibly resolve conflicts. We describe the design of PANE's operators and its use of the HFT compiler in §5.

Having summarized PANE's key ideas, we now describe at a high level the processing of a single request, as depicted in Figure 1. When an authenticated principal sends the controller a message, perhaps requesting a resource for a flowgroup in a particular share, PANE first checks that the request is admissible per the share's flowgroup and privileges – Diamond 1 in the figure.

If this first check passes, PANE then checks to see if it is compatible with the state of the network – Diamond 2. This check involves all accepted requests (*i.e.*, policy atoms) in the policy tree, and the physical capabilities of the network. For example, a bandwidth reservation requires a circuit between two endpoints with sufficient bandwidth and switch queues. This check requires compiling the current policy tree, augmented with the request itself. If this check passes (*i.e.*, if the request is feasible), the request is incorporated into the tree, and the controller can install the policy onto the network. This process also has a variation which only partially fulfills requests; §5.2 describes both variations in more detail.

A final key feature, which we detail in subsequent sections, is that PANE allows principals to request resources for future intervals. To support this, PANE maintains a time-indexed sequence of policy trees. The above checks may thus be made against future, planned network state as appropriate.

3. INTERACTING WITH PANE

We now expand upon the three message types introduced in the overview: requests, queries, and hints. Table 1 has a concise specification of these messages, and their relation to other key concepts in PANE's API.

3.1 Requests

A *request* affects the state of the network for some interval of time. By default, requests take effect immediately and do not expire; this allows critical network invariants to be expressed easily. Specific start and end times may optionally be provided. Verifying if a request can be granted may require walking the tree's hierarchy, depending on the type of request. This design allows resources to be oversubscribed; overallocation is prevented when requests are granted, and not when shares are created.

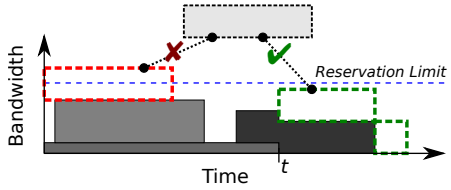


Figure 2: Example user request for reserved bandwidth; PANE determines that it cannot be fulfilled until time t.

Participatory networks may support requests for a variety of network resources and services, which we detail next.

Access Control The simplest type of network service exposed by PANE is access control – the ability to allow and deny traffic, using the **Allow** and **Deny** requests. Like all requests, they specify a flowgroup describing the affected traffic, and the share which the principal is using to invoke the privilege. Each access control privilege is optionally constrained by a specified number of seconds, n. To exceed this limit, principals must periodically renew requests. Shares lacking the ability to allow or deny traffic have $n = 0$. When creating a sub-share, a principal cannot exceed these constraints. For example, if a share carries the privilege to **Deny** traffic for up to 300 seconds, a sub-share cannot be created with the privilege to **Deny** traffic for up to 301 seconds.

The handling of a given packet is ultimately decided by the composition of every matching access control request. This composition makes use of the share tree's hierarchy to resolve conflicts – for example, an access control request made on a child share overrides those in parent shares. We defer further discussion of PANE's general approach to conflict resolution until §5.

With each request, the principal can specify a fulfillment mode, either *strict* or *partial*. These are provided for atomicity and convenience. In strict mode, PANE rejects a request if it would be (partially) overridden by any previous request. For example, if a user wants to allow connections to TCP ports 1000-2000, but there exists a request in a sub-share that denies port 1024, PANE rejects the request, explaining why. In partial mode, PANE implements the request, and informs the user that it was only partially satisfied; in the same example, PANE would inform the user that it has allowed ports 1000-1023, and 1025-2000.

These modes exist for two reasons: first, to avoid race conditions in request allocations, and second, to avoid complicated, fine-grained specifications that depend on PANE's current state. We defer a more complete discussion of the strict and partial fulfillment modes until §5.2.

Guaranteed Minimum Bandwidth PANE also provides a **Reserve** privilege which provides guaranteed minimum bandwidth (GMB) between two hosts. Shares which contain the privilege to reserve bandwidth are limited by a modified token bucket: it has the usual attributes of fill rate F, capacity C, and maximum drain rate M, and an additional minimum drain rate m. This lower bound prevents reservations with very low drain rates that could last indefinitely. A simple reservation with maximum bandwidth B is a special case with $F = M = B; C = m = 0$. GMB reservations are ultimately implemented by PANE's runtime as a sequence of forwarding actions and switch queues, as we describe in §6. Requests which cannot be implemented are rejected.

Figure 2 shows a simple example in which a principal has requested an immediate bandwidth reservation. PANE determines that granting the request will exceed the share's available bandwidth. The principal then examines the share's schedule of available band-

Share	S	$\in \{P\} \times \{F\} \times \{Priv\}$	A share gives principals some privileges to affect a set of flows.
Principal	P	::= (**user**, *host*, **app**)	A triple consisting of an *application*, running on a *host* by a *user*.
Flow	F	::= $\langle$**srcIP**=n_1, **dstIP**=n_2, **proto**=n_3, **srcPort**=n_4, **dstPort**=$n_5\rangle$	A set of packets with shared properties: source and destination IP address, transport protocol, and source and destination transport ports.
Privilege	*Priv*	::= **CanDeny** n | **CanAllow** n	The privileges to allow or deny traffic for up to n seconds (optional).
		| **CanReserve** n | **CanRateLimit** n	The privileges to reserve bandwidth or set rate-limits, up to n MB.
		| **CanWaypoint** $\{IP\}$ | **CanAvoid** $\{IP\}$	The privileges to direct traffic through or around particular IP addresses.
Message	*Msg*	::= $P : \{F\} : S \rightarrow$ (*Req Tspec* | *Hint Tspec* | *Query*)	A message from a principal with a request, hint, or query using a share.
Time Spec	*Tspec*	::= **from** t_1 **until** t_2	An optional specification from time t_1 until t_2.
Request	*Req*	::= **Allow** | **Deny**	Request to allow/deny traffic.
		| **Reserve** n | **RateLimit** n	Request to reserve n MB or rate-limit to n MB.
		| **Waypoint** *IP* | **Avoid** *IP*	Waypoint/avoid traffic through a middlebox with the given IP address.
Query	*Query*	::= **TrafficBetween** *srcIP dstIP* | ...	Query the total traffic between two hosts.
Hint	*Hint*	::= **Duration** t | ...	Hint that the flow's duration is t.
Policy Atom	*Atom*	::= $P : \{F\} \rightarrow$ *Req Tspec*	A requested modification of network state.
		| **Hint** $P : \{F\} \rightarrow$ *Req Tspec*	A realized hint; it may be removed if it conflicts with a future request.

Table 1: Main concepts in PANE

width and sends a new request for a reservation to start at t; PANE accepts the request and later implements it.

When creating sub-shares of shares with GMB privileges, the sub-share's token bucket must "fit inside" the parent's token bucket; parents cannot provide more tokens to their children than they receive. However, a share's tokens can be over-subscribed by its sub-shares. Over-subscription with sub-shares allows a principal to delegate access to all available bandwidth in a share more flexibly than by delegating access directly. By creating sub-shares, PANE's extensible conflict resolution (§5) can mediate between the child shares' requests. To prevent over-allocation, PANE draws tokens from all of its parent shares, up to the root of the tree when a request is granted.

Path Control A third request type directs flows through or around middleboxes using **Waypoint** and **Avoid**. For example, a university's network administrators can route students' traffic through a packet shaper during business hours, and security researchers can avoid intrusion detection systems for traffic to be collected by honeypots. Shares contain sets of IP addresses listing the middleboxes which they can route through or avoid, and, as with flowgroups, sub-shares may only contain subsets of their parents' sets. PANE implements **Waypoint** and **Avoid** by installing flow-specific forwarding rules along a path determined by fixing or deleting nodes as appropriate when routing over the network graph (§6.1). Requests to create unrealizable paths are rejected.

Rate-limits PANE can support rate-limit requests which result in matching traffic being routed through ports with established rate-limiters, as available in current switches. While basic, such requests can be used to mitigate DoS attacks or enforce traffic contracts between tenants in a shared datacenter. PANE's global view of the network enables it to make best use of the switches' features and place rate-limiters close to the traffic's source, as we describe in §6.1. Like PANE's bandwidth reservations, rate-limits are currently restricted to circuits; a network with distributed rate-limiters, such as those proposed by Raghavan, *et al.* [39], could support more general limits, and their use could be integrated into PANE as well.

3.2 Queries

PANE also supports messages to query the state of the network. These queries may be for general information about the network, such as the type of a link (*e.g.*, copper or optical), the set of hosts located downstream of a particular port, or other properties. Each share may contain a list of properties which it is privileged to read. This list is similar to a "view" on a database; when sub-shares are created, this view may be further occluded. While these restrictions provide basic privacy protection when exposing the network's state, they are not complete. For example, if a switch has three links, and a principal has the privilege to read the sending and receiving rates on two of the links, but not the third, it can infer the rate on the third link. We leave a more complete development of privacy protections as future work.

The current OpenFlow specifications and design make a number of properties available which principals in PANE may query including: the number (or list) of hosts behind a particular port, port-specific diagnostic data such as the number of packets dropped, the number of CRC errors, etc., the physical and topological location of switches, and the access medium of links. In the future, we would like to support additional details we believe would benefit applications such as the current signal-to-noise ratio or broadcasting power of wireless access points.

PANE also supports a "network weather service" which provides coarse information about current traffic conditions. For example, statistics about the total traffic over inter-switch links are available, but not statistics about individual flows. Support for collecting such detailed statistics requires a more robust OpenFlow compiler (*e.g.*, Frenetic's [19]) than the one in PANE's current implementation.

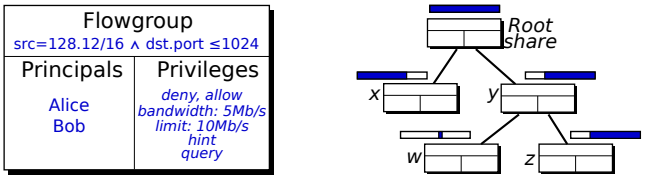
Applications can issue queries to the PANE controller to improve the user experience. For example, Hadoop could use the weather service to place reducers away from currently-congested parts of the network, and streaming video players can determine that a wireless access point is attached to a cellular modem or similarly constrained backhaul as Shieh, *et al.* proposed [42].

3.3 Hints

The final type of message in PANE is a hint. Hints are used to provide the network with information which may improve the application's or network's performance, without creating an additional requirement. Providing hints across abstraction boundaries is a natural feature in other systems.

Three hints which are useful for networked applications include: a flow's size in bytes, a desired flow-completion deadline, and the predictability of future traffic. PANE can use flow size information to spread large flows across multiple paths of equal-cost, as in Mahout [12] or Hedera [1]. Deadlines can be communicated to supporting routers such as those proposed in D3 [48]. Hints about traffic predictability can be used by optimizers such MicroTE [5].

PANE *may* use hints to derive and install policy atoms which affect related traffic, although it gives no guarantee or notification to the user. For example, a hint that a flow is short may generate



(a) (b)

Figure 3: (a) A PANE share. (b) A share hierarchy. The rectangle above each share represents a flowgroup according to one dimension (*e.g.*, source IP). Sub-shares are defined on a subset of their parent's flowgroup, and may not have more permissive privileges than their parent.

a policy atom to increase that flow's priority. We call such hints *realized*, and their corresponding policy atoms are tagged as merely hints (cf. Table 1).

The integration of hints, which can benefit non-PANE systems, as in the examples above, is deliberate. PANE provides a network administrator with the framework to delegate privileges, divide resources, and account for their usage; the ability to issue hints is a privilege, particularly those which affect limited resources. The framework provided by PANE makes it more feasible to implement hints in an untrusted environment, where malicious principals may issue false or excessive hints in an attempt to gain an advantage.

Finally, in the absence of transactional-style requests (*e.g.*, a request for "resource A *or* resource B"), PANE's hints are a more flexible way to provide information to the network than via requests. In this use, hints share a similar role to PANE's partial fulfillment mode for requests (§5.2).

4. PRIVILEGE DELEGATION

This section presents the semantics of shares and how principals' messages are authorized in more detail. The PANE controller maintains two key data structures. First, the *share tree* determines the privileges that principals have to read or write network state. The tree-structure allows principals to create new shares and delegate authority to each other. The share tree itself does not affect the state of the network. Instead, the second key data-structure, the *policy tree*, holds *policy atoms* that can affect the network. PANE maintains the invariant that all policy atoms in the policy tree are properly authorized by the share tree at all times.

A share-tree is an n-ary tree of *shares*, where a share gives a set of *principals* some *privileges* to affect a set of *flows* in the network. We elaborate on these terms below.

Principals A PANE principal is a triple consisting of an application running on a host by a user. For example, a principal may be (**Skype**, *192.168.1.7*, **Alice**) or (**Hadoop**, *10.20.20.20*, **Bob**). Shares in PANE are held by principal-sets. We abbreviate singleton sets to their principal. We also use wildcards to denote large sets. *e.g.*, (**Alice**, $\star$, $\star$) is the set of all principals with Alice as the user, and ($\star$, $\star$, **Hadoop**) is the set of all principals with Hadoop as the application. We write ($\star$, $\star$, $\star$) to denote the set of all principals.

Principals send messages to the PANE controller to request resources and query the state of the network. For example, the principal (**Skype**, *192.168.1.7*, **Alice**) may request low-latency service between the Skype call's source and destination, and the principal (**Hadoop**, *10.20.20.20*, **Bob**) may request guaranteed bandwidth between the three machines in an HDFS write pipeline, as we implement in §7.1.

In a deployed system, PANE could use 802.1x to authenticate the user portion of a principal against an existing user database such as Active Directory or LDAP. In an ideal environment, the application and host portions could be attested to by a TPM module and application signatures on the end host [43]. For now, our prototype only considers the user portion of a principal.

The three-part principal design allows users and network administrators to fully understand the provenance of each request. For example, in a cluster of Hadoop machines, requests by different Application Masters are identifiable back to the specific machine they were made from. Similarly, users can differentiate between requests from distinct applications on the same machine.

Flows A flow is a set of related packets on which requests are made. For example,

$$\langle \textbf{srcIP}{=}w, \textbf{dstIP}{=}x, \textbf{proto}{=}\mathit{TCP}, \textbf{srcPort}{=}y, \textbf{dstPort}{=}z \rangle$$

is a flowgroup that denotes a TCP connection from $w : y$ to $x : z$. A PANE share allows principals to affect a set of flows, which we denote with wildcards when possible. For example, the following flowgroup denotes all HTTP requests:

$$\langle \textbf{srcIP}{=}\star, \textbf{dstIP}{=}\star, \textbf{proto}{=}\mathit{TCP}, \textbf{srcPort}{=}\star, \textbf{dstPort}{=}80 \rangle$$

whereas the following denotes HTTP requests and responses:

$$\langle \textbf{srcIP}{=}\star, \textbf{dstIP}{=}\star, \textbf{proto}{=}\mathit{TCP}, \textbf{srcPort}{=}\star, \textbf{dstPort}{=}80 \rangle \cup$$
$$\langle \textbf{srcIP}{=}\star, \textbf{dstIP}{=}\star, \textbf{proto}{=}\mathit{TCP}, \textbf{srcPort}{=}80, \textbf{dstPort}{=}\star \rangle$$

A key invariant of the share tree is that if share S_1 is a sub-share of share S_2, then S_1's flowgroup is a subset of S_2's flowgroup. Therefore, sub-shares allow principals to implement fine-grained delegation of control.

Privileges Privileges in PANE define the messages principals may send using the share. Each message type, as described in the previous section, has a corresponding privilege. For example, **CanAllow** n and **CanDeny** n permit admission-control policies to be requested for n seconds, and **CanWaypoint** $\{IP\}$ indicates that principals can route traffic through an IP address in the given set.

5. CONFLICT RESOLUTION

Conflicts arise naturally in a participatory network, as PANE is designed to allow multiple, distributed principals to author the network configuration. For example, one principal may issue a request to deny traffic to TCP port 80, while another may request such traffic be allowed. This section discusses how PANE handles conflicts between overlapping requests.

Two requests overlap when the intersection of their respective flowgroups is not empty, *i.e.*, there are some flows that match both. As described in §2, principals make requests in the context of a share, and accepted requests become policy atoms residing in this share. Policy atoms, then, inherit from the share tree a natural hierarchical relationship, which we call the *policy tree*. The network's effective policy is a function of the set of all policy atoms, their position in the tree, and the semantics of conflict resolution between overlapping policy atoms.

The semantics of the policy tree is the final action it produces on an individual packet, after it has consolidated the actions of all policy atoms in the tree. Figure 4 illustrates a packet's evaluation: matching policy atoms (shown in green) produce an action, such as **Allow** (shown in blue), and conflicts are resolved up the hierarchy until a final action is emitted from the tree.

The policy tree is a declarative representation of the effective policy implemented by PANE and installed in the physical network. In

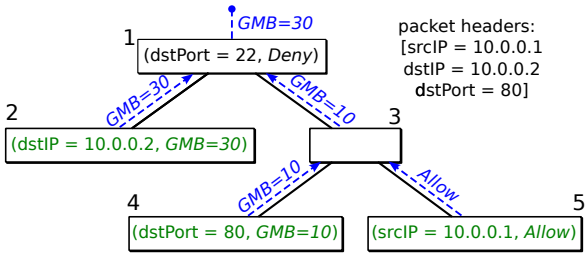


Figure 4: Evaluation of a single packet

PANE, we represent policy trees using HFTs, or Hierarchical Flow Tables [16]. HFTs are a natural choice for PANE as they provide two key features: first, flexible conflict resolution through the use of *conflict resolution operators*; and second, a formally-verified compiler from HFTs to the flow match tables used in OpenFlow. HFT's *eval* function implements the evaluation strategy described above. We now describe how PANE uses the operators to resolve conflicts (§5.1), the compiler to strictly or partially fulfill requests (§5.2), and the complexity of this approach (§5.3).

5.1 Conflict-resolution Operators

HFTs resolve conflicts through the use of conflict resolution operators. These operators take two conflicting requests as input, and return a single resolved request. For example, a packet which matches policy atoms from **Reserve**(10) and **Reserve**(30) may be resolved to the higher guaranteed bandwidth, **Reserve**(30), as occurs at Node 1 in Figure 4.

HFTs have three types of conflict-resolution operators at each node in the tree. These multiple types allow HFTs to resolve different types of conflicts using independent logic: $+_D$ is used to resolve conflicts between requests within the same share, $+_P$ between conflicting requests in parent and child shares, and $+_S$ to resolve conflicts between sibling shares. Their placement directly in the nodes allows conflict resolution to make implicit use of the hierarchy. This design makes it simple to express intuitive conflict resolutions such as "child overrides parent."

For PANE we chose simple conflict-resolution operators in the interest of user and administrator understanding. PANE's parent-child operator ($+_P$) specifies a "child overrides parent" policy for admission control. PANE's $+_S$ and $+_D$ operators are identical, and specify a "**Deny** overrides **Allow** policy" between siblings.

5.2 Strict vs Partial Fulfillment

We now return to PANE's *strict* and *partial* modes of fulfillment, first introduced with the **Allow** and **Deny** privileges. In each mode, a request is first authenticated against the share tree, then, as shown in Figure 1, PANE verifies the resulting policy tree can be compiled to a valid network configuration. After this verification, the two modes differ.

In strict mode, PANE ensures that a request's specified action is the same as the action returned by HFT's *eval* function for all packets in the request's flowgroup – that is, no conflict resolution operator has changed the resulting action for any matching packets. More formally, when a request with match rule M and action A is added to a policy tree, yielding tree T, $\forall$ packets $K \in \{K | M \cap K \neq \emptyset\}$, $eval(T, K) = A$. If this condition does not hold, the request is rejected. In partial mode, the request is not subject to this check, and may even be relaxed – for example, a request for 30 Mbps of guaranteed bandwidth on a share with only 20 Mbps available will be relaxed to a request for 20 Mbps.

These modes are useful for three reasons. First, strict mode provides the principal with a guarantee that the request will be implemented in the network as specified. This is a limited form of change-impact analysis: *was the impact of my change on the network's configuration what I expected? If not, cancel the request.* We will expand PANE's ability to provide change-impact analysis in future work.

Second, partial mode improves support for concurrent requests, as at least a relaxed form of a partial request will succeed. Without this, a principal faces the risk of repeatedly crafting strict requests based on the network state at time t_0, only to have the request arrive at time $t_2 > t_0$ and conflict with a request accepted at time t_1, where $t_2 > t_1 > t_0$.

Finally, partial mode's ability to relax a request is a useful convenience. For example, if a principal has permissions which affect dozens of specific TCP ports in the range 1000-2000, yet not all of them, partial requests can be made for that range, and the requests would be relaxed to just the specific ports, freeing the principal from needing to specify the particular ports on each request.

Partial reservations, such as the 20 Mbps received of the 30 Mbps requested in the example above, are particularly useful as applications can use them to provide upper-bounds for transfer time. Although the faster reservation may have been preferred, the slower one still provides predictability to the end-user (and in either scenario, the actual bandwidth received by the transfer may be even higher than the guaranteed minimum). Such a use case is different from that for bandwidth hints; with hints, the principal does not know how the information will be used, if at all.

5.3 Compiler Complexity

To realize a policy tree in OpenFlow hardware, we have to compile it to flow tables for each switch. We use a variation of Hierarchical Flow Tables (HFT) [16]. A direct implementation of the HFT algorithm produces flow tables of size $O(2^n)$, where n is the size of the policy tree. This algorithm is therefore useless on all but trivial policies. However, we make two changes that greatly reduce the complexity: the modified algorithm yields flow tables of size $O(n^2)$ in $O(n^2)$ time. This section is an overview of our results.

OpenFlow flow tables are simple linear sequences of patterns and actions. A flow can match several, overlapping policy atoms in a policy tree and trigger conflict-resolution that combines their policies. However, in an OpenFlow flow table, a flow will only trigger the action of the highest-priority matching pattern.

For example, suppose the policy tree has two atoms with the following flowgroups:

$\langle$**srcIP**=X, **dstIP**=Y, **proto**=`tcp`, **srcPort**=$\star$, **dstPort**=$\star\rangle$
$\langle$**srcIP**=$\star$, **dstIP**=$\star$, **proto**=`tcp`, **srcPort**=$\star$, **dstPort**=`80`$\rangle$

Suppose flows that match the first flowgroup – all flows from X to Y – are waypointed through some switch, and that flows that match the second flowgroup – all HTTP requests – are given some bandwidth reservation. These two flowgroups overlap, thus a flow may be (1) waypointed with a reservation, (2) only waypointed, (3) only given a reservation, or (4) not be affected by the policy.

An OpenFlow flow table that realizes the above two-atom policy tree must have entries for all four cases. The original algorithm [16] generates all possible combinations given trees of size n — *i.e.* flow tables of size $O(2^n)$.

We make two changes to prune the generated flow table: (1) we remove all rules that generate empty patterns and (2) we remove all rules whose patterns are fully shadowed by higher-priority rules. The earlier algorithm is recursive, and we prune after each recursive call. It is obvious that this simple pruning does not affect the

semantics of flow tables. However, a surprising result is that it dramatically improves the complexity of the algorithm.

The intuition behind our proof is that for sufficiently large policy trees, the intersections are guaranteed to produce duplicate and empty patterns that get pruned. To see this, note OpenFlow patterns have a bit-vector that determines which fields are wildcards. If patterns have h header fields, there are only 2^h unique wildcard bit-vectors. Therefore, if a policy tree has more than 2^h policy atoms, then by the pigeonhole principle some patterns must have identical wildcard bits.

Consider two policy atoms with the same wildcard bits. If the two patterns are identical, then so is their intersection. Therefore, the original two patterns get pruned, leaving only the intersection. Now, suppose the two patterns are distinct (and still have the same wildcard bits). Therefore, since their wildcards are the same, they both match some header differently, and thus their intersection is empty and pruned.

Our full complexity analysis, available in an extended tech report [17], shows that when the number of policy atoms, n, is larger than 2^h, then the compilation algorithm runs in $O(n^2)$ time and produces a flow table of size $O(n^2)$. Note that h is effectively a constant, fixed by the number of header fields which may determine a flow; this value is limited by the number of fields defined in OpenFlow. OpenFlow 1.0 patterns are 12-tuples, and our current policies only use 5 header fields. Therefore, on policies with more than 2^5 policy atoms, the algorithm is quadratic.

Updating Flow Tables

It is not enough for PANE to generate flow tables quickly. It must also propagate switch updates quickly, as the time required to update the network affects the effective duration of requests. The OpenFlow protocol only allows switches to be updated one rule at a time. A naive strategy is to first delete all old rules, and then install new rules. In PANE, we implement a faster strategy: the controller state stores the rules deployed on each switch; to install new rules, it calculates a "diff" between the new and old rules. These diffs are typically small, since rule-table updates occur when a subset of policy atoms are realized or unrealized.

6. THE PANE CONTROLLER

The complete PANE system integrates the previously described components into a fully-functioning SDN controller, as depicted in Figure 1. It manages simultaneous connections with the network's principals and its switches. In this role, it is responsible for implementing both our participatory networking API, and the details of computing default forwarding routes, transmitting OpenFlow messages, and reacting to network changes such as switches joining and links failing. To accomplish these tasks, the PANE controller maintains three data structures: the share tree, a sequence of policy trees, and a *network information base* (NIB), described below.

We have developed a prototype PANE controller using Haskell and the Nettle library for OpenFlow [46]. We use and extend the HFT compiler described in [16]. Although we chose OpenFlow as our substrate for implementing PANE, its design does not depend on OpenFlow. PANE could be implemented using other mechanisms to control the network, such as 4D [24], MPLS, or a collection of middleboxes.

The PANE controller is an entirely event-driven multicore program. The three primary event types are incoming PANE API messages, incoming OpenFlow messages, and timer events triggered by the start or finish of previously accepted requests or realizable hints. A prototype release is available on Github, and we provide a virtual machine for Mininet-based evaluation on our website.[1] The release also includes a Java library which implements an object-oriented interface to PANE's text API.

API messages always specify a share on which they are operating. When a message arrives, the PANE controller first uses the share tree to determine whether it is authorized, and then, for requests, whether it is feasible by consulting the policy trees, as described in the previous sections.

When requests start and expire, the PANE controller compiles the new policy tree to a set of switch flow tables, translating high-level actions to low-level operations on individual switches in the network. For example, a **Reserve**(n) action becomes a circuit of switch queues and forwarding rules that direct packets to those queues. As we will describe next, PANE's runtime uses its NIB and a default forwarding algorithm to realize this and other actions. Our implementation constructs a spanning tree and implements MAC learning as its forwarding algorithm.

When possible, PANE uses the slicing extension to OpenFlow 1.0 to create queues, and out-of-band commands when necessary. While OpenFlow allows us to set expiry timeouts on flow table entries, PANE must explicitly delete queues when reservations expire.

6.1 Network Information Base

A network information base (NIB) is a database of network elements – hosts, switches, ports, queues, and links – and their capabilities (*e.g.*, rate-limiters or per-port output queues on a switch). The runtime uses the NIB to translate logical actions to a physical configuration, determine a spanning tree for default packet forwarding, and to hold switch information such as manufacturer, version, and its ports' speeds, configurations, and statistics.

For example, PANE's runtime implements a bandwidth reservation, $(M, \textbf{Reserve}(n))$, by querying the NIB for the shortest path with available queues between the corresponding hosts. Along this path, PANE creates queues which guarantee bandwidth n, and flow table rules to direct packets matching M to those queues. We chose this greedy approach to reserving bandwidth for simplicity, and leave the implementation of alternatives as future work.

PANE also uses the NIB to install **Deny** rules as close as possible to the traffic source. For example, if the source is outside our network, this is the network's gateway switch. If the source is inside the network, packets are dropped at the closest switch(es) with available rule space. The NIB we implement is inspired by Onix [30]. It uses a simple discovery protocol to find links between switches, and information from our forwarding algorithm, such as ARP requests, to discover the locations of hosts.

6.2 Fault Tolerance and Resilience

The PANE controller must consider two types of failures. The first is failure of network elements, such as switches or links, and the second is failure of the controller itself.

When a switch or link fails, or when a link's configuration changes, the PANE runtime must recompile the policy tree to new individual switch flow tables, as previously used paths may no longer be available or acceptable. Because the underlying network has changed, this recompilation step is not guaranteed to succeed. If this happens, we defer to PANE's first come-first serve service model, greedily replaying requests to build a new policy tree which does compile; implementing this simply requires annotating the current policy tree's policy atoms with the order in which they were created. Principals are notified via call-backs if a previously accepted request is now unsatisfiable. Developing a more sophisticated approach to re-constructing a feasible policy tree, perhaps taking ad-

[1] `http://pane.cs.brown.edu`

vantage of priorities, or with the goal of maximizing the number of restored requests, remains as future work.

To handle failure of the controller, we can keep a database-like persistent redo log of accepted requests, periodically compacted by removing those which have expired. Upon recovery, the PANE controller could restore its state from this log. In production settings, we expect the PANE controller to be deployed on multiple servers with shared, distributed state. Switches would maintain connections to each of the controllers as newer OpenFlow specifications support. We leave the design and analysis of both options as future work. Because network principals use PANE in an opt-in fashion to receive predictable performance, a complete runtime failure would simply return the network to its current state of providing best-effort performance only.

6.3 Additional Features

The PANE runtime supports several additional features beyond the requests, hints, and queries previously described. Principals are able to query PANE to determine their available capabilities, examine the schedule of bandwidth availability, create sub-shares, and grant privileges to other principals. PANE's API also provides commands to determine which existing requests and shares can affect a specified flowgroup; this is particularly useful for debugging the network, such as to determine why certain traffic is being denied.

Beyond the API, the PANE controller also includes an administrative interface which displays the current state and configuration of the network, real-time information about the controller's performance such as memory and CPU usage, and allows the dynamic adjustment of logging verbosity.

7. EVALUATION

We evaluate our PANE prototype with the Mininet platform for emulating SDNs [32], and with real networks. Our primary testbed includes two Pronto 3290 switches and several software OpenFlow switches (both Open vSwitch and the reference user-mode switch) on Linux Intel-compatible hardware, and on the TP-Link WR-1043ND wireless router. Wired connections are 1 Gbps and wireless runs over 802.11n. Clients on the network include dedicated Linux servers, and fluctuating numbers of personal laptops and phones. In addition to the participatory networking API, the network also provides standard services such as DHCP, DNS, and NAT.

Members of our group have been using the testbed since February 2012 to manage our traffic, and during this time, it has been our primary source of network connectivity. The testbed is compatible with unmodified consumer electronic devices, which can interact with a PANE controller running at a well-known location.[2]

In the following sections, we examine two aspects of our prototype. First, we consider four case studies of real applications that use the PANE API to improve end-user experience (§7.1). Second, we evaluate the practicality of implementing the PANE API in current OpenFlow-enabled networks, considering questions such as the latency of processing requests, and the number of rules created by networked applications (§7.2).

7.1 Application Usage

We ported four real applications to use the PANE API: Ekiga, SSHGuard, ZooKeeper, and Hadoop. We now describe how intentions of an application developer or user can be translated to our API, and the effects of using PANE on the network and the application. Our PANE-enabled versions of these applications are all publicly available on Github.[3]

7.1.1 Ekiga

Ekiga is an open source video conferencing application. We modified Ekiga to ask the user for the anticipated duration of video calls, and use a **Reserve** message to request guaranteed bandwidth from the network between the caller's host and either the network gateway or the recipient's host, for the appropriate time. If such a reservation is not available, Ekiga retrieves the schedule of available bandwidth from PANE and calculates the earliest time at which a video call or, alternatively, an audio call, can be made with guaranteed quality. It then presents these options to the user, along with a third option for placing a "best effort" call right away.

Realizable reservations cause the PANE controller to create guaranteed bandwidth queues along the path of the circuit, and install forwarding rules for Ekiga's traffic.

Measurements of Skype use on a campus network with more than 7000 hosts show that making reservations with PANE for VoIP applications is quite feasible. Skype calls peaked at 75 per hour, with 80% of calls lasting for fewer than 30 minutes [6]. This frequency is well within current OpenFlow switches' capabilities, as we measure in §7.2.

7.1.2 SSHGuard

SSHGuard is a popular tool to detect brute-force attacks via log monitoring and install local firewall rules (*e.g.*, via `iptables`) in response. We modified SSHGuard to use PANE as a firewall backend to block nefarious traffic entering the network. In particular, this means such traffic no longer traverses the targeted host's access link.

For example, if Alice is running SSHGuard on her host and it detects a Linux syslog entry such as:

```
sshd[2197]: Invalid user Eve from 10.0.0.3
```

SSHGuard will block Eve's traffic for the next five minutes using PANE's **Deny** request. The PANE controller then places an OpenFlow rule to drop packets to Alice's host coming from Eve's at a switch close to Eve's host.

Although this is a basic example, it illustrates PANE's ability to expose in-network functionality (namely, dropping packets) to end-user applications. Besides off-loading work from the end-host's network stack, this approach also protects any innocent traffic which might have suffered due to sharing a network link with a denial-of-service (DoS) attack.

To demonstrate this benefit, we generated a UDP-based DoS attack within our testbed network. We started an `iperf` TCP transfer between two wireless clients, measured initially at 24 Mbps. We then launched the attack from a Linux server two switch-hops away from the wireless clients. During the attack, which was directed at one of the clients, the performance of the `iperf` transfer dropped to 5 Mbps, rising to only 8 Mbps after the victim installed a local firewall rule. By using PANE to block the attack, the transfer's full bandwidth returned.

7.1.3 ZooKeeper

ZooKeeper [27] is a coordination service for distributed systems used by Twitter, Netflix, and Yahoo!, among others, and is a key component of HBase. Like other coordination services such as Paxos [31], ZooKeeper provides consistent, available, and shared

[2]The PANE controller could also be specified using a DHCP vendor-specific or site-specific option.

[3]`https://github.com/brownsys/`

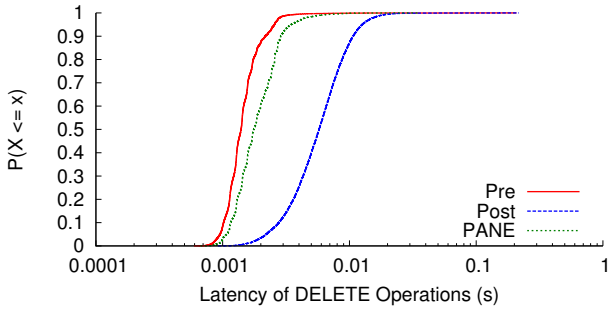


Figure 5: Latency of ZooKeeper DELETE requests.

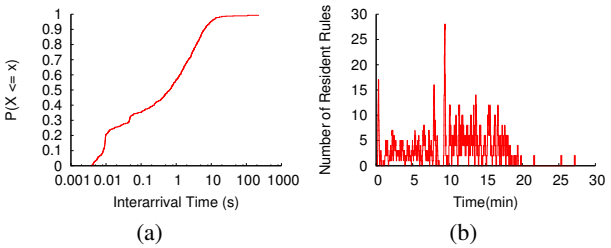


Figure 6: Effect of Hadoop on PANE and network.

state using a quorum of replicated servers (the *ensemble*). For resiliency in the face of network failures, ZooKeeper servers may be distributed throughout a datacenter, and thus quorum messages may be negatively affected by heavy traffic on shared links. Because ZooKeeper's role is to provide coordination for other services, such negative effects are undesirable.

To protect ZooKeeper's messages from heavy traffic on shared links, we modified ZooKeeper to make bandwidth reservations using PANE. Upon startup, each member of the ensemble made a reservation for 10 Mbps of guaranteed minimum bandwidth for messages with other ZooKeeper servers. Additionally, we modified our ZooKeeper client to make a similar reservation with each server it connected to.

We installed ZooKeeper on an ensemble of five servers, and developed a benchmarking client which we ran on a sixth. The client connected a thread to each server and maximized the throughput of synchronous ZooKeeper operations in our ensemble. To remove the effect of disk latency, the ZooKeeper servers used RAM disks for storage. At no time during these experiments were the CPUs of the client, switches, or servers fully loaded. Like our modified applications, this benchmarking tool is also available on Github.

Figure 5 shows the latency of ZooKeeper DELETE requests during the experiment. In the "Pre" line, ZooKeeper alone is running in the network and no reservations were made using PANE. In the "Post" line, we used `iperf` to generate bi-directional TCP flows over each of the six links directly connected to a host. As shown in the figure, this competing traffic dramatically reduced ZooKeeper's performance – average latency quadrupled from 1.55ms to 6.46ms (we obtained similar results with a non-OpenFlow switch). Finally, the "PANE" line shows the return to high performance when ZooKeeper reserved bandwidth using PANE.

We found similar results for other ZooKeeper write operations such as creating keys, writing to unique keys, and writing to the same key. Read operations do not require a quorum's participation, and thus are less affected by competing background traffic.

7.1.4 Hadoop

In our final case study of PANE's application performance benefits, we augmented a Hadoop 2.0.3 pre-release with support for our API. Hadoop is an open source implementation of the MapReduce [13] data-processing framework. In Hadoop, large files are divided across multiple nodes in the network, and computations consist of two phases: a map, and a reduce. During the map phase, a function is evaluated in parallel on independent file pieces. During the reduce, a second function proceeds in parallel on the collected outputs of the map phrase; the data transfer from the mappers to the reducers is known as the shuffle. During the shuffle, every reduce node initiates a transfer with every map node, making it particularly network-intensive for some jobs, such as sorts or joins. Finally, the output of the reduce is written back to the distributed filesystem.

By using PANE, our version of Hadoop is able to reserve guaranteed bandwidth for its operations. The first set of reservations occurs during the shuffle – each reducer reserves bandwidth for transferring data from the mappers. The second set of reservations reserves bandwidth when writing the final output. These few reservations protect the majority of network transfers that occur during the lifetime of a Hadoop job. Our version of Hadoop also makes reservations when a map task needs to read its input across the network; however, such transfers are typically less common thanks to "delay scheduling" [52]. Therefore, in a typical job, the total number of reservations is on the order of $M \times R + R \times 2$ where M and R are, respectively, the number of nodes with map and reduce tasks. The number of reservations is not precisely described by this formula as we do not make reservations for node-local transfers, and reducers may contact a mapper node more than once during the shuffle phase. As reducers can either be copying output from mappers in the shuffle, or writing their output to the distributed filesystem, the maximum number of reservations per reducer at any time is set by the value of *mapreduce.reduce.shuffle.parallelcopies* in the configuration, which has a default value of five.

To measure the effect of using PANE to make reservations in Hadoop, we developed a benchmark which executed three 40 GB sort jobs in parallel on a network of 22 machines (20 slaves, plus two masters) connected by a Pronto 3290 switched controlled by PANE. Hadoop currently has the ability to prioritize or weight jobs using the scheduler, but this control does not extend to the network. In our benchmark, the first two jobs were provided with 25% of the cluster's memory resources, and the third, acting as the "high priority" job, was provided with 50%. The benchmark was run in two configurations: in the first, Hadoop made no requests using PANE; in the second, our modified Hadoop requested guaranteed bandwidth for each large flow. These reservations were proportional to the job's memory resources, and lasted for eight seconds, based on Hadoop's 256 MB block size. In our star topology with uniform 1 Gbps links, this translated to 500 Mbps reservations for each link.

Averaged across three runs, the high priority job's completion time decreased by 19% when its bandwidth was guaranteed. Because it completed more quickly, the lower priority jobs' runtime also decreased, by an average of 9%, since Hadoop's work-conserving scheduler re-allocates freed memory resources to remaining jobs.

While Hadoop was running, we also measured its effect on PANE and the switch's flow table. Figure 6(a) is a CDF of the time between Hadoop's reservations. As currently implemented, PANE modifies the switch flow table after each request. This CDF shows that batching requests over a 10 ms window would decrease the number of flow table updates by 20%; a 100 ms window would decrease the updates by 35%. Figure 6(b) plots the amount of flow table space used by Hadoop during a single job. On average, Hadoop accounted

for an additional 2.5 flow table entries; the maximum number of simultaneous Hadoop rules was 28.

7.2 Implementation Practicality

In addition to examining PANE's use in real applications, we also evaluated the practicality of its implementation in current OpenFlow networks. We found that our Pronto 3290 switches, running the Indigo 2012.09.07 firmware, were capable of supporting 1,919 OpenFlow rules, which took an average of 7.12 ms to install per rule. To measure this, we developed a benchmarking controller which installed wildcard match rules, issuing a barrier request after each `flow_mod` message was sent. We based this controller on Floodlight, and it is available for download from our Github page.

The latency distribution to fully install each `flow_mod` is show in Figure 7(a). It has two clusters – for the 92.4% of `flow_mod`'s with latency less than 10.0 ms, the average latency was 2.80 ms; the remaining 7.6% had an average latency of 59.5 ms. For PANE's principals, these much higher tail latencies imply that requests cannot always be implemented within a few milliseconds, and for truly guaranteed traffic handling, requests have to be made at least 100 milliseconds in advance.

We found that our Pronto switches could support seven hardware queues with guaranteed minimum bandwidth on each port, and each queue required an average of 1.73 ms to create or delete, as shown in Figure 7(b). However, this average doubles to 3.56 ms if queues are created consecutively on the same port (*i.e.*, P1Q1, P1Q2, P1Q3, ..., P2Q1, etc.), as shown in Figure 7(c). This shows that an optimized PANE controller must consider the order in which switch operations are made to provide the best experience.

Together, these results suggest that an individual switch can support a minimum of about 200 reservations per second. Higher throughput is possible by batching additional requests between OpenFlow barriers. While these switch features are sufficient to support the four applications above, we found that Hadoop's performance benefited from per-flow reservations only when flows transferred more than one megabyte. For smaller flows, the overhead of establishing the reservations outweighed the benefit. In an example word count job, only 24% of flows were greater than 1 MB; however this percentage rises to 73% for an example sort.

8. DISCUSSION AND FUTURE WORK

The PANE prototype implements a complete OpenFlow controller: it determines network-wide policies by realizing requests and hints, monitors switches to respond to queries, determines routing, and updates switches as the policy changes. We've built each of these components as separate modules, and compose them using composition abstractions derived from Frenetic [19].

In the PANE prototype, policies are arranged as trees. This is not a fundamental design decision – trees make it easy to implement resource accounting and trace the provenance of delegation decisions. However, generalizations such as DAGs should be possible.

An interesting generalization is a more flexible language for requests. A principal should be able to describe the resources they need as simple constraints. For example, "reserve 5 Gbps for 1 hour within the next 5 hours." Such requests would give the PANE controller more scheduling flexibility. They would thus improve network performance and lead to greater accepted requests.

We are also investigating allowing principals to request other kinds of resources, *e.g.* latency and jitter. Current OpenFlow implementations lack support for controlling these traffic properties. In the future, we hope to borrow techniques from works such as Borrowed Virtual Time [14] or HFSCs [44], which integrate and balance the needs of throughput- and latency-sensitive processes.

Security Considerations While the principals in PANE are authenticated, they do not need to be trusted as privileges are restricted by the system's semantics – the ShareTree restricts the capabilities of each principal. We recognize, however, that it may be possible to exploit combinations of privileges in an untoward fashion, and leave such prevention as future work.

Our prototype implementation of PANE is currently defenseless against principals which issue excessive requests. We leave such protection against denial-of-service as future work, and expect PANE's requirement for authenticated principals to enable such protections, as use of the system can be audited.

PANE as a building block We believe the primitives provided by PANE could serve as an implementation substrate for a variety of recent proposals in the networking literature.

For example, Coflow [11] proposes an abstraction which groups several related flows, allowing the network control-plane to better optimize an application's communication. By adding support for transactions, we expect PANE could be used to implement Coflow – each coflow would map to a sequence of PANE requests grouped by a transaction. FairCloud [38], which develops several policies for fairly dividing datacenter bandwidth, should be implementable using PANE's reservation and rate-limit requests.

Finally, because PANE allows applications to reserve guaranteed bandwidth, such applications could skip TCP's slow start phase, or even allow for the network control-plane to be involved in setting congestion control and other parameters, as in proposals such as XCP [28] and OpenTCP [21]. PANE could also integrate better support for middlebox-based services, perhaps by integrating the approach advocated for by Gember, *et al.* [20].

9. RELATED WORK

Programming the Network PANE allows applications and users to influence network operations, a goal shared by previous research such as active networking [45]. In active networks, principals develop distributed programs that run in the network nodes. By contrast, PANE sidesteps active networks' deployment challenges via its implementation as an SDN controller, their security concerns by providing a much more restricted interface to the network, and their complexity by providing a logically centralized network view.

Using Application-Layer Information Many previous works describe specific cases in which information from end-users or applications benefits network configuration, flexibility, or performance; PANE can be a unifying framework for these. For example, using Hedera to dynamically identify and place large flows in a datacenter can improve throughput up to 113% [1]. PANE avoids Hedera's inference of flow size by enabling applications and devices to directly inform the network about flow sizes. Wang, *et al.* [47] propose *application-aware networking*, and argue that distributed applications can benefit from communicating their preferences to the network control-plane, as we show in §7.1. ident++ [35] proposes an architecture in which an OpenFlow controller reactively queries the endpoints of a new flow to determine whether it should be admitted. TVA is a network architecture in which end-hosts authorize the receipt of packet flows via capabilities in order to prevent DoS-attacks [50]. By contrast, PANE allows administrators to delegate the privilege to install restricted network-wide firewall rules, and users can do so either proactively or reactively (cf. §7.1.2).

Darwin [9] introduced a method for applications to request network resources, including computation and storage capabilities in network processors. Like PANE, Darwin accounts for resource use hierarchically. However, it does not support over-subscription, lacks

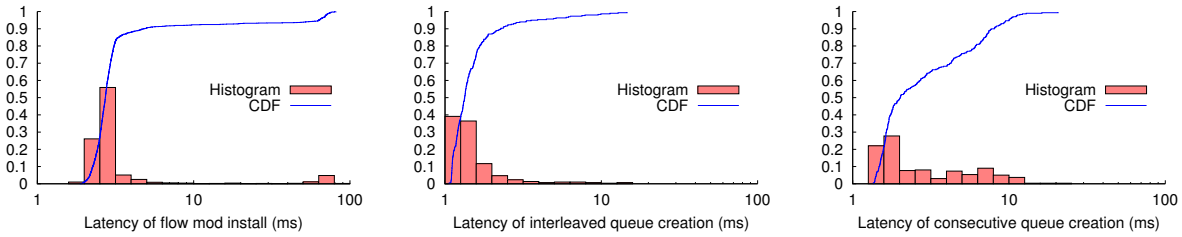


Figure 7: Latency of switch operations in milliseconds.

support for access control and path management, and requires routers with support for active networks. Yap, *et al.* have also advocated for an explicit communication channel between applications and software-defined networks, in what they called *software-friendly networks* [51]. This earlier work, however, only supports requests made by a single, trusted application. By contrast, PANE's approach to delegation, accounting, and conflict-resolution allow multiple applications to safely communicate with an SDN controller.

Network QoS and Reservations Providing a predictable network experience is not a new goal, and there is a vast body of protocols and literature on this topic. PANE relies heavily on existing mechanisms, such as reservations and prioritized queue management [29, 44], while adding user-level management and resource arbitration. PANE also goes beyond QoS, integrating hints and guarantees about access control and path selection. To date, we have focused on mechanisms exposed by OpenFlow switches; we expect other mechanisms for network QoS could be integrated as well.

Like PANE, protocols such as RSVP [8] and NSIS [33] provide applications with a way to reserve network resources on the network. PANE, however, is designed for single administrative domains, which permits centralized control for policy decisions and accounting, and sidesteps many of their deployment difficulties. PANE provides control over the configuration of network paths, which RSVP and NSIS do not, and goes beyond reservations with its hints, queries, and access control requests, which can be made instantly or for a future time. Finally, RSVP limits aggregation support to multicast sessions, unlike PANE's support for flow groups.

Kim, et al. [29] describe an OpenFlow controller which configures QoS using application-described requirements and a database of network state. PANE's runtime performs a similar function for the **Reserve** action, and also supports additional actions.

Recent works in datacenter networks, such as Oktopus [3] and CloudNaaS [4], offer a predictable experience to tenants willing to fully describe their needs as a virtual network, only admitting those tenants and networks whose needs can be met through careful placement. This approach is complementary to PANE's, which allows principals to request resources from an existing network without requiring complete specification.

Software-Defined Networking PANE is part of a line of research into centralized network management including Onix [30], Tesseract [49], and CoolAid [10]. CoolAid provides high-level requests and intentions about the network's configuration to its operators; PANE extends this functionality to regular users and applications with the necessary delegation and accounting, and implements them in SDNs. PANE builds upon the abstractions proposed by Onix and Tesseract for, respectively, OpenFlow and 4D [24] control-planes.

Recent developments in making SDNs practical (*e.g.*,[25, 34, 46]) improve the deployability of PANE. Resonance [36] delegates access control to an automated monitoring system, using OpenFlow to enforce policy decisions. Resonance could be adapted to use PANE as the mechanism for taking action on the network, or could be composed with PANE using a library such as Frenetic [19].

Expressing policies in a hierarchy is a natural and common way to represent delegation of authority and support distributed authorship. Cinder [40], for example, uses a hierarchy of *taps* to provide isolation, delegation, and division of the right to consume a mobile device's energy. PANE uses HFTs [16] as a natural way to express, store, and manipulate these policies directly, and still enable an efficient, equivalent linear representation of the policy.

FlowVisor [41] divides a single network into multiple slices independently controlled by separate OpenFlow controllers. FlowVisor supports delegation – a controller can re-slice its slice of the network. Each of these controllers sends and receives primitive OpenFlow messages. In contrast, PANE allows policy authors to state high-level, declarative policies with flexible conflict resolution.

Networking and Declarative Languages PANE's design is inspired by projects such as the Margrave tool for firewall analysis [37] and the Router Configuration Checker [15], which apply declarative languages to network configuration. Both use a high-level language to detect configuration mistakes in network policies by checking against predefined constraints. PANE, however, directly integrates such logic into the network controller.

FML [26] is a Datalog-inspired language for writing policies that also supports distributed authorship. The actions in PANE are inspired by FML, which it extends by involving end-users, adding queries and hints, and introducing a time dimension to action requests. In an FML policy, conflicts are resolved by a fixed scheme – deny overrides waypoints, and waypoints override allow. By contrast, PANE offers more flexible conflict resolution operators. FML also allows policies to be prioritized in a linear sequence (a *policy cascade*). PANE can also express a prioritized sequence of policies, in addition to more general hierarchies.

The eXtensible Access Control Markup Language (XACML) provides four combiner functions to resolve conflicts between subpolicies [23]. These functions are designed for access control decisions and assume an ordering over the subpolicies. By contrast, HFTs support user-supplied operators designed for several actions and consider all children equal.

10. CONCLUSION

The design and configuration of today's networks is already informed by application needs (*e.g.*, networks with full-bisection bandwidth for MapReduce-type frameworks, or deadline-based queuing [3] for interactive web services). PANE provides a way for the network to solicit and react to such needs automatically, dynamically, and at a finer timescale than with human input. To do this,

our design overcomes the two challenges of decomposing network control, and resolving conflicts between users' needs.

Acknowledgments

This work was partially supported by NSF grant 1012060. Andrew Ferguson is supported by an NDSEG fellowship. We thank Theo Benson, Srikanth Kandula, Joe Politz, Jennifer Rexford, Scott Shenker, and our shepherd Vyas Sekar for invaluable discussions and suggestions; Justin Pombrio for improving the implementation of PANE's Network Information Base; Jordan Place for first implementing PANE support in Ekiga; and Jeff Rasley for help with the Hadoop experiments.

11. REFERENCES

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI '10*.

[2] `https://aws.amazon.com/message/65648/`.

[3] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *SIGCOMM '11*.

[4] T. Benson, A. Akella, A. Shaikh, and S. Sahu. CloudNaaS: A Cloud Networking Platform for Enterprise Applications. In *SOCC '11*.

[5] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *CoNEXT '11*.

[6] D. Bonfiglio, M. Mellia, M. Meo, and D. Rossi. Detailed analysis of skype traffic. *IEEE Trans. on Multimedia*, 11(1):117–127, 2009.

[7] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC 1633, June 1994.

[8] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP). RFC 2205, Sept. 1997.

[9] P. Chandra, A. Fisher, C. Kosak, T. S. E. Ng, P. Steenkiste, E. Takashi, and H. Zhang. Darwin: Resource Management for Value-added Customizable Network Service. In *IEEE ICNP '98*.

[10] X. Chen, Y. Mao, Z. M. Mao, and J. Van der Merwe. Declarative Configuration Management for Complex and Dynamic Networks. In *CoNEXT '10*.

[11] M. Chowdhury and I. Stoica. Coflow: An Application Layer Abstraction for Cluster Networking. In *HotNets '12*.

[12] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-Overhead Datacenter Traffic Management using End-Host-Based Elephant Detection. In *IEEE INFOCOM '11*.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[14] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *SOSP '99*.

[15] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI '05*.

[16] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Hierarchical Policies for Software Defined Networks. In *HotSDN '12*.

[17] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. Technical Report CS-13-03, Brown Univ., 2013.

[18] A. D. Ferguson, A. Guha, J. Place, R. Fonseca, and S. Krishnamurthi. Participatory Networking. In *Hot-ICE '12*.

[19] N. Foster, M. J. Freedman, R. Harrison, J. Rexford, M. L. Meola, and D. Walker. Frenetic: A High-Level Language for OpenFlow Networks. In *PRESTO '10*.

[20] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Towards Software-Defined Middlebox Networking. In *HotNets '12*.

[21] M. Ghobadi, S. H. Yeganeh, and Y. Ganjali. Rethinking End-to-End Congestion Control in Software-Defined Networks. In *HotNets '12*.

[22] `https://github.com/blog/1346-network-problems-last-friday`.

[23] S. Godik and T. M. (editors). eXtensible Access Control Markup Language, version 1.1, Aug. 2003.

[24] 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. *SIGCOMM CCR*, 35:41–54, 2005.

[25] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM CCR*, 38:105–110, July 2008.

[26] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical Declarative Network Management. In *WREN '09*.

[27] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait free coordination for Internet-scale systems. In *USENIX ATC '10*.

[28] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM '02*.

[29] W. Kim, P. Sharma, J. Lee, S. Banerjee, J. Tourrilhes, S.-J. Lee, and P. Yalagandula. Automated and Scalable QoS Control for Network Convergence. In *INM/WREN '10*.

[30] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI '10*.

[31] L. Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[32] B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *HotNets '10*.

[33] J. Manner, G. Karagiannis, and A. McDonald. NSIS Signaling Layer Protocol (NSLP) for Quality-of-Service Signaling. RFC 5974, Oct. 2010.

[34] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR*, 38:69–74, 2008.

[35] J. Naous, R. Stutsman, D. Mazières, N. McKeown, and N. Zeldovich. Enabling delegation with more information. In *WREN '09*.

[36] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: dynamic access control for enterprise networks. In *WREN '09*.

[37] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The Margrave tool for firewall analysis. In *LISA '10*.

[38] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurth, S. Ratnasamy, and I. Stoica. FairCloud: Sharing The Network In Cloud Computing. In *SIGCOMM '12*.

[39] B. Raghavan, K. V. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren. Cloud Control with Distributed Rate Limiting. In *SIGCOMM '07*.

[40] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. Energy Management in Mobile Devices with the Cinder Operating System. In *EuroSys '11*.

[41] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the Production Network Be the Testbed? In *OSDI '10*.

[42] A. Shieh, E. G. Sirer, and F. B. Schneider. Netquery: A Knowledge Plane For Reasoning About Network Properties. In *SIGCOMM '11*.

[43] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical Attestation: An Authorization Architecture For Trustworthy Computing. In *SOSP '11*.

[44] I. Stoica, H. Zhang, and T. S. E. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *SIGCOMM '97*.

[45] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. Wetherall, and G. Minden. A Survey of Active Network Research. In *IEEE Communications Magazine*, January 1997.

[46] A. Voellmy and P. Hudak. Nettle: Taking the Sting Out of Programming Network Routers. In *PADL '11*.

[47] G. Wang, T. S. E. Ng, and A. Shaikh. Programming Your Network at Run-time for Big Data Applications. In *HotSDN '12*.

[48] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better never than late: meeting deadlines in datacenter networks. In *SIGCOMM '11*.

[49] H. Yan, D. A. Maltz, T. S. E. Ng, H. Gogineni, H. Zhang, and Z. Cai. Tesseract: A 4D Network Control Plane. In *NSDI '07*.

[50] Z. Yang, D. Wetherall, and T. Anderson. A DoS-limiting Network Architecture. In *SIGCOMM '05*.

[51] Yap, Kok-Kiong and Huang, Te-Yuan and Dodson, Ben and Lam, Monica S. and McKeown, Nick. Towards Software-Friendly Networks. In *APSys '10*.

[52] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys '10*.