Hierarchical Policies for Software Defined Networks

Andrew D. Ferguson
Brown University
adf@cs.brown.edu

Arjun Guha
Brown University
arjun@cs.brown.edu

Chen Liang
Brown University
chen_liang@cs.brown.edu

Rodrigo Fonseca
Brown University
rfonseca@cs.brown.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

ABSTRACT

Hierarchical policies are useful in many contexts in which resources are shared among multiple entities. Such policies can easily express the delegation of authority and the resolution of conflicts, which arise naturally when decision-making is decentralized. Conceptually, a hierarchical policy could be used to manage network resources, but commodity switches, which match packets using flow tables, do not realize hierarchies directly.

This paper presents *Hierarchical Flow Tables* (HFT), a framework for specifying and realizing hierarchical policies in software defined networks. HFT policies are organized as trees, where each component of the tree can independently determine the action to take on each packet. When independent parts of the tree arrive at conflicting decisions, HFT resolves conflicts with user-defined conflict-resolution operators, which exist at each node of the tree. We present a compiler that realizes HFT policies on a distributed network of OpenFlow switches, and prove its correctness using the Coq proof assistant. We then evaluate the use of HFT to improve performance of networked applications.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*network management*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

General Terms

Management, Design, Experimentation

Keywords

Software Defined Networks, OpenFlow, Participatory Networking, hierarchical policies

1. INTRODUCTION

Hierarchies are useful in many contexts in which resources are shared and delegated among multiple entities. For example, Linux cgroups [1] organize processes in trees to express and control the sharing of memory, CPU, and I/O bandwidth, and the Cinder operating system [13] uses a hierarchy to control the sharing of battery resources among smartphone applications. In general, hierarchies are a common representation of delegation and accounting for shared resources.

This makes hierarchies a natural way to delegate management of network resources. For example, a network administrator may wish to delegate the authority to block external traffic to a security team. The team leader may wish to further delegate this authority to deputies, and if two deputies disagree on whether to block a particular flow, the leader may want a conservative conflict-resolution policy, such as "deny overrides allow." In a campus network, the network administrator may delegate authority for traffic engineering to several teams, such as media services for simulcasting, or scientists for bulk data transfers.

Existing systems, such as the previously mentioned cgroups and Cinder, can natively use a tree structure for accounting and delegation. Although software defined networks enable custom packet-handling at each switch, we cannot install hierarchical policies directly on commodity hardware.

This paper presents the design, realization, and evaluation of hierarchical policies in software defined networks. We develop *Hierarchical Flow Tables* (HFT), which provide a semantics for policy trees – a representation of hierarchical policies designed for networks (§2). We also develop a compiler and runtime system to realize policy trees on a network of OpenFlow switches [10] by integrating information about the network topology (§3).

Finally, HFT enables *Participatory Networking* [2], a framework in which end-users and their applications propose changes to the network configuration. With this, we evaluate the use of HFT to improve the performance of networked applications in our prototype participatory network (§5). Existing systems [3, 5, 11, 16], which we compare in §6, do not support hierarchical network-wide policies such as ours.

2. SEMANTICS OF HFT

HFT allows several principals to author a tree of policies, and specify custom conflict-resolution operators at each node in the tree. In this section, we define the semantics of a policy tree as the final action it produces on an individual packet, after it has consolidated actions from all policies in the tree.[1] In §3, we compile these policy trees to run efficiently on hardware.

Figure 1 defines packets (K), policy trees (T), actions (A), and a function $eval$ that matches packets against policy trees and returns an action. For our purposes, packets are a vector of header names

[1] This semantic model, where the central controller conceptually sees all packets, is inspired by Frenetic [3].

HotSDN'12, August 13, 2012, Helsinki, Finland.

$$\begin{array}{lrcl}
 & H & = & \text{header names and ingress ports} \\
\text{patterns} & V & = & \textit{const} \mid \textit{prefix} \mid \star \\
\text{matches} & M & = & \emptyset \mid \langle \overrightarrow{H,V} \rangle \\
\text{actions} & A & = & \textbf{Allow} \mid \textbf{Deny} \mid \textbf{GMB}(n) \mid \mathbf{0} \\
\text{conflict-resolution} & (+) & = & A \to A \to A \\
\text{operators} & & & \\
\text{policy atoms} & P & = & M \times A \\
\text{policy tree nodes} & D & = & (+_D) \times 2^P \\
\text{policy trees} & T & = & (+_P) \times (+_S) \times D \times 2^T \\
\text{packets} & K & = & \langle \overrightarrow{H, \textit{const}} \rangle
\end{array}$$

$$\boxed{cmb : D \times K \to A}$$

$$\begin{array}{l}
cmb((+, \{\cdots(M_i, A_i)\cdots\}), K) = A'_1 + \cdots + A'_k + \mathbf{0} \\
\quad \text{where} \quad \{A'_1, \cdots, A'_k\} \quad = \quad \{A_i | M_i \cap K \neq \emptyset\}
\end{array}$$

$$\boxed{eval : T \times K \to A}$$

$$\begin{array}{l}
eval((+_P, +_S, D, \{T_1, \cdots, T_n\}), K) = cmb(D, K) +_P A_1 \\
\quad \text{where} \quad \begin{array}{rcl} A_1 & = & eval(T_1, K) +_S A_2 \\ A_2 & = & eval(T_2, K) +_S A_3 \\ \cdots & & \\ A_n & = & eval(T_n, K) +_S \mathbf{0} \end{array}
\end{array}$$

Figure 1: Semantics of HFT[2]

and values; we do not match on packets' contents. For concreteness, we depict the actions we have implemented in our prototype (§3): admission control, guaranteed minimum bandwidth (GMB), and **0**, a special "don't care" action. In §7, we outline how to support additional ones such as rate-limiting and waypointing.

A policy tree is a tree of policy nodes (D), which contain sets of policy atoms (P). An atom is a match rule and action pair, (M, A). When a packet matches a policy atom, $M \cap K \neq \emptyset$, the atom produces its action. The interesting cases occur when a packet matches several policy atoms with conflicting actions. In these cases, we resolve conflicts with the conflict-resolution operators $(+)$ attached throughout the policy tree.

Policy trees have different types of conflict-resolution operators at several points in the tree (*i.e.*, $+_D$, $+_P$, $+_S$ in Figure 1). These multiple types allow HFT to resolve different types of conflicts using independent logic. For example, conflicts between parent and child nodes may be resolved differently than conflicts between a single node's internal policy atoms. Therefore, the choice of conflict-resolution operators is a key policy decision. Our prototype network (§4) provides two default operators; developing and evaluating additional operators is left as future work.

The function cmb matches a packet with an individual policy tree node. If a packet matches several policy atoms, cmb uses the node's internal conflict-resolution operator, $+_D$, to combine their actions. The compiler requires $+_D$ to be associative, commutative, and have **0** as its identity.[3]

The function $eval$ matches a packet with a policy tree by applying cmb to the policy tree node at the root, and recursively applying $eval$ to its children. A policy tree has conflict-resolution operators $+_P$ and $+_S$, which respectively allow it to resolve parent-child and inter-sibling conflicts differently. In particular, $+_P$ does not have to be commutative – it is always used with the parent's action on the left and the child's action on the right. This lets us express intuitive conflict resolutions such as "child overrides parent."

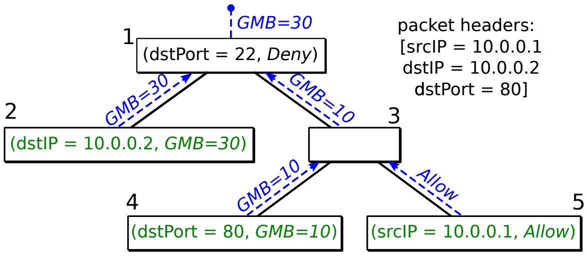

Figure 2: Evaluation of a single packet

Example: Figure 2 depicts a simple policy tree and illustrates how $eval$ produces an action, given the tree and indicated packet. Each node contains its policy atoms, and atoms which match the packet are colored green. The $eval$ function recursively produces an action from each sub-tree; these actions are the labels on each node's outgoing edge.

In this example, the policy atoms at each leaf match the packet and produce an action. Node 3 receives conflicting actions from its children, which it resolves with its inter-sibling conflict-resolution operator: $\textbf{GMB}(10) +_S \textbf{Allow} = \textbf{GMB}(10)$. Node 3 has no policy atoms itself, so it produces the **0** action. Since **0** is the identity of all conflict-resolution operators, $\mathbf{0} +_P \textbf{GMB}(10) = \textbf{GMB}(10)$ is the resulting action from this sub-tree.

Finally, Node 1 computes the aggregate action of its children: $\textbf{GMB}(30) +_S \textbf{GMB}(10) = \textbf{GMB}(\max(30, 10))$. Since Node 1's policy atoms do not match the packet, the final action is $\mathbf{0} +_P \textbf{GMB}(30) = \textbf{GMB}(30)$.

3. COMPILING POLICIES

The preceding section assumes that a central function, $eval$, observes and directs all packets in the network. Although $eval$ specifies the meaning of policy trees, this is not a practical implementation. We now describe how to compile HFT's policy trees to run on commodity switches, which support simpler, linear flow tables, to produce a practical implementation.

Our compiler works in two stages. First, we translate policy trees to *network flow tables*, which have a basic, linear matching semantics (§3.1). Second, we use network flow tables to configure a distributed network of switches, translating high-level actions such as $\textbf{GMB}(n)$ to low-level operations on switches (§3.2).

3.1 Network Flow Tables

A network flow table (N) is a sequence of paired match rules and actions. The $scan$ function, defined in Figure 3, matches packets against network flow tables and returns the action associated with the first matching rule. If no rules match the packet, then $scan$ returns **0**.[4]

The matching semantics of network flow tables correspond to the matching semantics of switch flow tables exposed by OpenFlow. When a packet matches a switch flow table, only one rule's action applies. If a packet matches multiple rules, the switch selects the one with the highest priority. A rule's index in a network flow table corresponds to a switch flow table priority, with index 0 as the highest priority. Since all rules have distinct indices, a naive correspondence would give all rules distinct priorities. A more compact

[2] $2^{M \times A}$ is the set of all subsets of pairs drawn from M and A.

[3] That is, we require $a + \mathbf{0} = \mathbf{0} + a = a$.

[4] The $scan$ function is derived from NetCore [11].

$$\text{Network Flow Tables} \quad N \quad = \quad \langle \overrightarrow{M, A} \rangle$$

$$\boxed{scan : N \times K \to A}$$

$$\frac{M_1 \cap K = \emptyset \cdots M_{j-1} \cap K = \emptyset \qquad M_j \cap K \neq \emptyset}{scan(\langle (M_1, A_1) \cdots (M_n, A_n) \rangle, K) = A_j}$$

$$\frac{M_1 \cap K = \emptyset \cdots M_n \cap K = \emptyset}{scan(\langle (M_1, A_1) \cdots (M_n, A_n) \rangle, K) = \mathbf{0}}$$

$$\boxed{lin_D : D \to N}$$

$$\begin{aligned} lin_D\,(+_D, \{M_1, A_1, \cdots, M_j, A_j\}) &= N_1 \\ \text{where} \quad N_1 &= union(+_D, \langle M_1, A_1 \rangle, N_2) \\ &\cdots \\ N_j &= union(+_D, \langle M_j, A_j \rangle, \langle\rangle) \end{aligned}$$

$$\boxed{lin_T : T \to N}$$

$$\begin{aligned} lin_T\,(+_P, +_S, D, \{T_1 \cdots T_k\}) &= union(+_P, lin_D(D), N_1) \\ \text{where} \quad N_1 &= union(+_S, lin_T(T_1), N_2') \\ &\cdots \\ N_k &= union(+_S, lin_T(T_k), \langle\rangle) \end{aligned}$$

$$\boxed{union, inter : (+) \times N \times N \to N}$$

$$\begin{aligned} &union((+), N_1, N_2) = inter((+), N_1, N_2) N_1 N_2 \\ &inter((+), \langle \cdots (M_i, A_i) \cdots \rangle, \langle \cdots (M_j', A_j') \cdots \rangle) = \\ &\quad \langle \cdots (M_i \cap M_j', A_i + A_j')) \cdots \rangle \end{aligned}$$

Figure 3: Network Flow Tables

one, which we use, maps a sequence of non-overlapping network flow table rules to a single priority in a switch flow table.

The lin_T function is our compiler from policy trees to network flow tables. It uses lin_D as a helper to compile policy tree nodes. The lin_D function translates policy atoms to singleton network flow tables, and combines them with $union(+, N, N')$. *Union* builds a network flow table that matches packets in either N or N'. Moreover, when a packet matches both N and N', *union* computes the intersection using the $+$ conflict-resolution operator to combine actions.

Similarly, lin_T recursively builds network flow tables for its subtrees, and calls lin_D on its root node. It applies *union* to combine the results, using $+_S$ and $+_P$ where appropriate.

The functions in Figure 3, lin_T, lin_D, *union*, and *inter* require the conflict-resolution operators to satisfy the following properties.

DEFINITION 1 (WELL-FORMED). *T is well-formed if:*

- *The $+_D$ and $+_S$ operators are commutative,*
- *All conflict-resolution operators are associative, and*
- ***0** is the identity of all conflict-resolution operators.*

Proving the compiler correct requires the following key lemma, which states that all conflict-resolution operators distribute over $scan$.

LEMMA 1. *For all $+$, N_1, and N_2, where **0** is the identity of $+$, $scan(union(+, N_1, N_2)) = scan(N_1) + scan(N_2)$.*

With this, we prove the compiler correct.

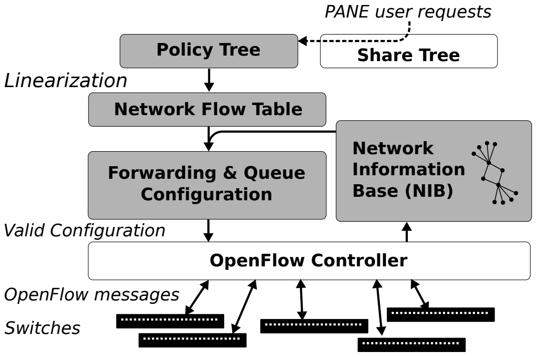

Figure 4: The PANE System

THEOREM 1 (SOUNDNESS). *For all well-formed policy trees, T and packets, P, $eval(T, P) = scan(lin_T(T), P)$.*

We mechanize all our definitions and proofs using the Coq proof assistant [15]. ■

3.2 The HFT Runtime

The final step in realizing policy trees is to translate the high-level actions in a network flow table to low-level operations on a network of switches. For example, a **GMB**(n) action needs to produce a circuit of switch queues and forwarding rules that direct packets to those queues. The HFT runtime uses a *network information base (NIB)* and a default forwarding algorithm to realize this and other actions.

A 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. For example, to implement a bandwidth reservation, $(M, \textbf{GMB}(n))$, the runtime queries the NIB for the shortest path with available queues between the corresponding hosts. Along this path, the runtime creates queues which guarantee bandwidth n, and flow table rules to direct packets matching M to those queues. We also use the NIB to install **Deny** rules as close as possible to the traffic source. The HFT runtime currently implements a greedy algorithm for both of these operations. Evaluating the efficiency of this approach and considering alternatives remains as future work.

The NIB we implement is inspired by Onix [8]. It uses a simple discovery protocol to find links between switches, and information from our forwarding algorithm to discover the locations of hosts. When possible, we use 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, our controller must explicitly delete queues when reservations expire.

As the network flow table does not specify packet forwarding, the runtime integrates decisions made by a separate forwarding algorithm. Our implementation uses MAC learning as its forwarding algorithm.

4. HFT IN PANE

HFT is the central component of PANE, our prototype implementation of Participatory Networking [2]. Figure 4 illustrates PANE's components, with HFT's components shaded gray. PANE allows users to submit requests for network resources with a start and end time. The PANE controller first checks that requests are authorized using a static share tree, which states which users can issue which

$$\boxed{+_P : A \times A \to A}$$

$$\begin{array}{llll}
A_P & +_P & \mathbf{0} & = A_P \\
\mathbf{0} & +_P & A_C & = A_C \\
\textbf{Deny} & +_P & \textbf{Allow} & = \textbf{Allow} \\
\textbf{Allow} & +_P & \textbf{Allow} & = \textbf{Allow} \\
A_P & +_P & \textbf{Deny} & = \textbf{Deny} \\
\textbf{Deny} & +_P & \textbf{GMB}(n) & = \textbf{GMB}(n) \\
\textbf{GMB}(m) & +_P & \textbf{GMB}(n) & = \textbf{GMB}(\max(m,n)) \\
\textbf{GMB}(m) & +_P & \textbf{Allow} & = \textbf{GMB}(m) \\
\textbf{Allow} & +_P & \textbf{GMB}(m) & = \textbf{GMB}(n)
\end{array}$$

$$\boxed{+_S, +_D : A \times A \to A}$$

$$\begin{array}{llll}
A_1 & +_S & \mathbf{0} & = A_1 \\
\mathbf{0} & +_S & A_2 & = A_2 \\
\textbf{Deny} & +_S & A_2 & = \textbf{Deny} \\
A_1 & +_S & \textbf{Deny} & = \textbf{Deny} \\
\textbf{GMB}(m) & +_S & \textbf{GMB}(n) & = \textbf{GMB}(\max(m,n)) \\
\textbf{GMB}(m) & +_S & \textbf{Allow} & = \textbf{GMB}(m) \\
\textbf{Allow} & +_S & \textbf{GMB}(m) & = \textbf{GMB}(m) \\
\textbf{Allow} & +_S & \textbf{Allow} & = \textbf{Allow} \\
A_1 & +_D & A_2 & = A_1 +_S A_2
\end{array}$$

Figure 5: PANE's conflict-resolution operators

requests about which flows in the network. It then verifies if the request can be realized in the network, adding it to its policy tree.

HFT's policy tree represents the accepted requests at a particular point in time. To check if an authorized request is realizable, PANE takes a snapshot of the policy tree with the potential request inserted at its start time. PANE then runs the HFT compiler on this policy tree to ensure that it will be realizable, that is, a valid network configuration can be built. If a valid configuration cannot be built – for example, if the NIB cannot find a circuit for a bandwidth reservation – PANE rejects the user's request.

When requests start and expire, the PANE controller sends OpenFlow messages constructed by HFT's runtime to affected switches. As Reitblatt, et al. [12] point out, it is a challenge to update switch configurations without introducing inconsistent, intermediate stages. Our implementation does not presently address this issue, but we anticipate confronting this problem in the future.

Figure 5 specifies PANE's conflict-resolution operators. The parent-child operator ($+_P$) specifies a "child overrides parent" policy for admission control. The $+_S$ and $+_D$ operators are identical, and specify a "**Deny** overrides **Allow** policy" between siblings. Our current implementation hard-codes these operators at all nodes, but we anticipate allowing principals to specify their own conflict-resolution operators.

5. EVALUATION

The true measure of the HFT runtime's success is its ability to successfully control traffic in an actual network. In particular, its control should reflect the integration of the NIB to provide virtual circuits with guaranteed minimum bandwidth, and optimize the placement of access control logic within the network. To capture this, we tested HFT as part of PANE, our prototype controller for a participatory network, and evaluated its performance in scenarios which exercised each benefit. In the first, a service reserved bandwidth to protect inter-process messaging, and in the second, an end-host used PANE to establish an in-network firewall to suppress a denial-of-service attack.

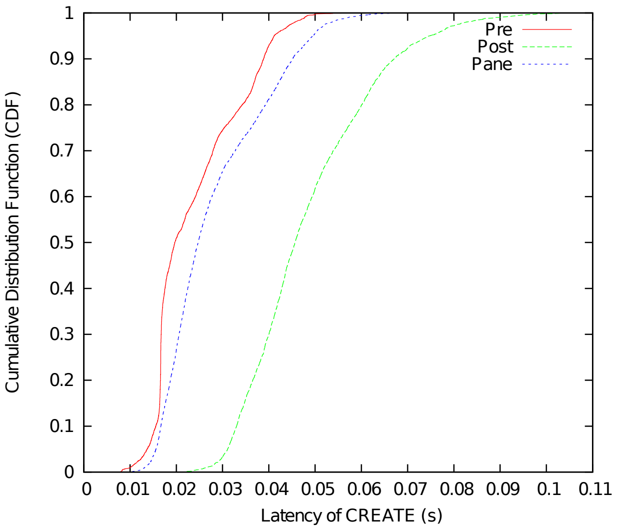

Figure 6: Latency of ZooKeeper CREATE requests.

The PANE testbed consists of software OpenFlow switches (both Open vSwitch and the reference user-mode switch) running on Linux Intel-compatible hardware and on the TP-Link WR-1043ND wireless router. Wired connections are 1 Gb/sec and wireless runs over 802.11n. Clients on the network include six dedicated Linux servers, plus fluctuating numbers of laptops and phones. The network also provides standard services such as DHCP, DNS, and NAT.

Members of our group have been using the testbed for more than four months 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 easily interact with a PANE controller running at a well-known location.[5]

5.1 ZooKeeper

ZooKeeper [6] is a coordination service for distributed systems used by Twitter, Netflix, and Yahoo! among others, and is key component of HBase. Like other coordination services such as Paxos [9], ZooKeeper provides consistent, available, and shared 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. At no time during these experiments were the CPUs of the client, switches, or servers fully loaded.

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

Figure 6 shows the latency of ZooKeeper CREATE requests during the experiment. In the "Pre" line, only ZooKeeper 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. This traffic totaled 3.3 Gbps, which we found to be the maximum Open vSwitch could sustain in our setup. As shown in the figure, this competing traffic dramatically reduced ZooKeeper's performance – average latency doubled from 24ms to 48ms (similar results were obtained with a non-OpenFlow switch). Finally, the "PANE" line shows ZooKeeper's return to high performance when it reserved bandwidth for its messages using PANE.

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

5.2 Thwarting Denial-of-Service

PANE provides firewall capabilities to participants using HFT's **Allow** and **Deny** actions. Because these rules are placed directly in the network by HFT's runtime, rather than on the requesting edge host, PANE can drop malicious traffic as it enters the network. This behavior 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. It was launched from a Linux server two switch-hops away from our wireless clients. Before the attack began, the clients were able to sustain a TCP transfer of 24 Mbps. During the attack, which was directed at one of the clients, the performance of this transfer dropped to 5 Mbps, rising only to 8 Mbps after the victim installed a local firewall rule. By using PANE to block the attack, the transfer's full bandwidth returned.

6. RELATED WORK

HFT gives policy writers a programming model in which a central controller sees all packets. This model is inspired by Frenetic [3], which uses NetCore [11] as its language for expressing forwarding policies. In contrast to NetCore, HFT supports *hierarchical policies*, which naturally support distributed authorship. A key element of HFT is allowing overlapping and conflicting policies to co-exist in a policy tree, as it resolves conflicts with arbitrary, user-defined operators.

NetCore allows policy writers to match packets with arbitrary predicates; when predicates are not realizable on switches, packets are sent to the NetCore controller, which uses *reactive specialization* to install exact-match rules. In contrast, HFT's matching rules are simpler and realizable on OpenFlow switches, though we do not anticipate any problem supporting reactive specialization.

FML [5] is a Datalog-inspired language for writing policies that also supports distributed authorship. In an FML policy, conflicts are resolved by a fixed scheme – deny overrides waypoints, and waypoints override allow. By contrast, HFT offers more flexibility, since each policy tree node can specify its own conflict resolution operator. For example, within a single HFT policy tree, one policy node may specify "allow overrides deny," while another specifies "deny overrides allow."

FML also allows policies to be prioritized in a linear sequence (a *policy cascade*), with higher-level policies overriding lower-level policies. HFT can express a prioritized sequence of policies, in addition to more general hierarchies. For example, PANE uses an inverted "child overrides parent" conflict-resolution scheme (§4) by default, but the author of an individual policy node can adopt a more restrictive "parent overrides child" scheme. FML does not support both "child overrides parent" and "parent overrides child" schemes simultaneously.

Nettle [16] is a platform for writing OpenFlow controllers in a functional reactive style. Our controller is built on Nettle, and allows policy authors to write higher-level policies than those Nettle supports natively.

FlowVisor [14] slices a single network so that several OpenFlow controllers can independently control each slice. 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, HFT allows policy authors to state high-level, declarative policies that are agnostic to network topologies and capabilities. HFT also permits policies to overlap and resolves conflicts using high-level conflict-resolution operators.

Kim, et al. [7] describe an OpenFlow controller which automatically configures QoS along flow paths using application-described requirements and a database of network state. HFT's runtime performs a similar function for the **GMB** action, but also supports additional actions.

TVA is a network architecture in which end-hosts authorize the receipt of packet flows via capabilities in order to prevent DoS-attacks [17]. HFT also supports this goal, but by allowing end-hosts to safely manage network policies for themselves, as we show in §5.2.

The eXtensible Access Control Markup Language (XACML) provides four combiner functions to resolve conflicts between subpolicies [4]. These functions are designed for access control decisions and assume an ordering over the subpolicies. By contrast, HFT supports user-supplied operators designed for several actions and considers all children equal.

7. FUTURE WORK

HFT is an extensible framework that can easily support additional actions beyond those described in Figure 1. For example, we could extend actions as follows:

$$\begin{array}{rcl} A & = & \cdots \mid \textbf{RateLimit}(n) \mid \textbf{Waypoint}(S) \mid \textbf{Avoid}(S) \\ S & = & \text{switch ID} \end{array}$$

A **RateLimit**(n) action would require the runtime to setup a rate-limiter on the switch closest to the source of traffic; our NIB can already determine the closest switch to implement **Deny** rules near the source. The **Waypoint**(S) and **Deny**(S), actions, which are inspired by FML [5], direct traffic to go through or avoid a particular switch; we could easily extend our NIB to find a path that must visit or avoid the given switch. We envision adding these and other high-level actions to HFT and compiling them to software defined networks with the HFT runtime.

The separation of policy trees from the underlying implementation presents them as an invariant which the runtime strives to maintain. As the network topology changes, the HFT runtime can adjust the policies' realization. For example, the addition of new switches may permit shorter paths. While we do not currently support such automatic reconfiguration in our prototype, we plan to do so in the future.

8. CONCLUSION

HFT enables hierarchical policies for software defined networks. These policies arise naturally when policy decisions are made by a decentralized group of authors, and can easily conflict with each other. Therefore, HFT's conflict-resolution operators, which authors can assign independently to each node, are a key part of an HFT policy.

HFT allows high-level, network-wide policies that do not require knowledge of the network topology. Its runtime uses a network information base (NIB), which provides topology information, such as shortest paths and nearest switches. This separation of high-level policies and topological details allows HFT to optimize the realization of its policies.

HFT makes Participatory Networking possible – end users and their applications can request high-level, abstract network resources, and rely on HFT to implement these requests. Our experiments show that HFT successfully compiles these requests to configurations which improve the performance of real applications.

Acknowledgments

This work was partially supported by NSF grant 1012060. Andrew Ferguson is supported by an NDSEG fellowship.

9. REFERENCES

[1] http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt. Last accessed April 6th, 2012.

[2] A. D. Ferguson, A. Guha, J. Place, R. Fonseca, and S. Krishnamurthi. Participatory Networking. In *Proc. Hot-ICE '12*, San Jose, CA, 2012.

[3] N. Foster, M. J. Freedman, R. Harrison, J. Rexford, M. L. Meola, and D. Walker. Frenetic: A High-Level Language for OpenFlow Networks. In *Proc. PRESTO '10*, Philadelphia, PA, 2010.

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

[5] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical Declarative Network Management. In *Proc. WREN '09*, Barcelona, Spain, 2009.

[6] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proc. USENIX ATC '10*, Boston, MA, 2010.

[7] 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 *Proc. INM/WREN '10*, San Jose, CA, 2010.

[8] 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 *Proc. OSDI '10*, Vancouver, BC, Canada, 2010.

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

[10] 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.

[11] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A Compiler and Run-time System for Network Programming Languages. In *Proc. POPL '12*, Philadelphia, PA, 2012.

[12] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent Updates for Software-Defined Networks: Change You Can Believe in! In *Proc. HotNets '11*, Cambridge, MA, 2011.

[13] 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 *Proc. EuroSys '11*, Salzburg, Austria, 2011.

[14] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the Production Network Be the Testbed? In *Proc. OSDI '10*, Vancouver, BC, Canada, 2010.

[15] The Coq Development Team. The Coq proof assistant reference manual – version 8.3. http://coq.inria.fr/, 2011.

[16] A. Voellmy and P. Hudak. Nettle: Taking the Sting Out of Programming Network Routers. In *Proc. PADL '11*, Austin, TX, 2011.

[17] Z. Yang, D. Wetherall, and T. Anderson. A DoS-limiting Network Architecture. In *Proc. SIGCOMM '05*, Philadelphia, PA, 2005.