Establishing Resource Allocation Policies in Federated Systems

Sapan Bhatia, Andy Bavier, Larry Peterson, Soner Sevinc
Princeton University

Abstract
Recent efforts to federate computation and communication resources across organizational boundaries face a challenge in establishing the policies by which the resources of one organization can be accessed by users from other organizations. To this end, this paper describes an approach to defining, communicating, analyzing, and enforcing resource allocation policies. Our approach was designed to address a specific need on PlanetLab, but we demonstrate through a wide range of examples that it is general enough to accommodate a diverse collection of computing facilities. Our approach is embodied in a specific tool chain, called sftables, that is patterned after the iptables mechanism used to define packet processing policies for network traffic. sftables provides a flexible interface for resource owners to specify access policies for their resources, and an underlying implementation that makes it possible to unambiguously communicate, query, and analyze allocation policies.

1 Introduction
Recent efforts to federate computation and communication resources across organizational boundaries face a challenge in establishing the policies by which the resources of one organization can be accessed by users of other organizations. This is true for federated network testbeds like PlanetLab [24] and GENI [13], as well as federated clouds like Open Cirrus [20]. The problem is how to give resource owners the ability to express access and allocation policies for their resources, along with the mechanisms needed to communicate, analyze, and enforce those policies.

Being able to control how users are able to access local resources is a barrier-to-entry for organizations wishing to contribute their resources to a global facility. Several PlanetLab-based testbeds—each owned and operated by an independent organization—illustrate some common concerns. Measurement-Lab (M-Lab) [18] and EmanicsLab [10] wish to limit their facilities to community-selected experiments that are consistent with the testbed’s purpose: supporting broadband measurement experiments and running network management experiments, respectively. Testbeds like VINI [4] (a layer-2 variant of PlanetLab deployed on the Internet2 and NLR backbones) want to do strict admission control so as to not overload their resources. In another case, G-Lab [12] (a VINI-like testbed deployed on the German research network) is restricted by funding sources to ensure that all German researchers are given sufficient resources before any international researchers can have access. A similar situation exists in OpenCirrus, where corporate policies and business relationships dictate what remote users are allowed to access each cluster. In all these cases—and others described throughout the paper—resource owners need a way to establish and enforce policies as to how their resources are allocated before they are willing to participate in a federated system.

Our experiences within the PlanetLab federation have suggested several high-level requirements that a resource allocation policy framework must satisfy:

- **Resource owners** must be able to express and enforce a wide range of policies across a rich set of physical and logical resources, as illustrated by the examples above.

- **Administrators** must be able to easily understand the framework and configure the desired policies.

- **Users** must be able to discover the policies that apply to them, so that they can take these policies into account when requesting resources.

A successful policy framework will balance the interests of all three parties.

This paper describes an approach to defining, communicating, analyzing, and enforcing resource allocation policies developed to support federation across PlanetLab, which in turn serves as a prototype of how federation might work in GENI. The approach is agnostic as to the type of facility being federated, offering a uniform framework for overlay networks, low-level network testbeds that have varied topologies, and cloud-like compute clusters.
Two design decisions underlie our approach. The first is to base our policy framework on the network firewall model. A network firewall defines a boundary, and implements policies about how users on one side of the boundary can access the network on the other side of the boundary. Our approach extends the idea of a firewall to encompass resources other than network access. Our policy engine, sfatables, sits “in front” of an organization’s resources, intercepting resource requests and accepting, denying, or modifying them like a firewall. Pushing the analogy further, sfatables leverages the same syntax and abstractions as the iptables firewall configuration utility; administrators use iptables-like rules operating on chains, matches, and targets to specify policies. Administrators familiar with iptables should be able to get up-to-speed quickly with sfatables.

The second design decision is to leverage the standardized XML tools in our policy framework; in particular, sfatables transforms policy rules into XSLT programs. In the PlanetLab federation, XML documents called RSpecs are used to request resources, so implementing policies using XML tools such as XSLT, a declarative language for transforming XML documents, seems a natural choice. However, basing sfatables on XSLT has three important benefits: (1) since XSLT is a declarative language, policies implemented in XSLT can be analyzed and queried by users, administrators, and other policy makers; (2) since XSLT is interpreted by browsers, XSLT programs are portable, allowing users to download and run policies off-line to help in crafting their resource requests; and (3) since XSLT is Turing-complete, sfatables can be extended to express and enforce arbitrary policies. XSLT policy programs help sfatables satisfy the requirements of both users and resource owners.

This paper makes three contributions. First, it proposes a general model for how resource usage policies might be established in federated systems. Our model is novel in that it treats the problem of establishing and enforcing resource allocation policies as orthogonal from other types of policy concerns (e.g., security, authentication). This model evolved out of our experiences addressing specific needs in PlanetLab, but as demonstrated by example policies, it is quite general. Second, it enumerates a collection of policies that have proved useful in PlanetLab. We posit them as examples of policies any federated system will need to support. Finally, it describes the design and implementation of sfatables. Our design offers administrators familiar syntax and abstractions, yet provides users and policy makers with a foundation for querying and analyzing an organization’s policies.

The paper is organized as follows. Section 2 provides an overview of the PlanetLab federation interfaces, and Section 3 presents our resource allocation model. Section 4 then gives some concrete examples showing how sfatables can be used to specify actual PlanetLab policies. Section 5 describes the design and implementation of sfatables. Section 6 explains how the XSLT policy programs at the core of sfatables can be transformed into Prolog constraint logic programs, providing a powerful policy analysis tool. Section 7 summarizes other approaches to establishing policies in federated environments.

2 Background

Federating resources contributed by autonomous organizations is a multi-faceted problem. It typically implies a common set of interfaces [2, 14, 9, 28], a shared identity management system [6, 15, 31, 8, 29, 30], and a distributed set of policies that define how all the federated elements play together. This paper is primarily concerned with the last of these three, and so as to focus on that topic in a concrete way, we describe our approach to establishing resource allocation policies in the context of a specific federated system—PlanetLab, which includes the “public” PlanetLab familiar to many researchers, as well as several “private” deployments that wish to federate with the public PlanetLab and with each other.

PlanetLab supports a set of interfaces—collectively called the Slice Federation Architecture (SFA)—that all organizations wishing to federate with the global system use to make their resources to the research community [25]. The full SFA includes operations for establishing identity, delegating rights, acquiring resources, and performing job control. For the purpose of this paper, however, we focus on two operations:

\[
\text{RSpec} = \text{LearnResources(Slice)} \\
\text{status} = \text{BindResources(Slice, RSpec)}
\]

both of which are exported by an aggregate—the abstraction representing a collection of computation and communication resources on behalf of some organization. Using the examples from the Introduction, M-Lab, EmanticsLab, VINI, and G-Lab all represent aggregates. Similarly, PlanetLab Europe (PLE) and PlanetLab Japan (PLJ) represent aggregates that correspond to regional subsets of the original unfederated PlanetLab; both are owned and managed by independent organizations that are distinct from PlanetLab Central.

Returning to the two relevant operations, the first returns the resource specification (RSpec) currently allocated to the named Slice. This RSpec consists of two
3 Resource Allocation Model

Our current work extends the resource allocation framework of the SFA to incorporate a Policy Engine that allows aggregate owners to specify access and allocation policies for their resources. The Policy Engine runs as part of the server that implements the SFA interface to the aggregate, and operates on the RSpec arguments of the LearnResources and BindResources calls. Running the Policy Engine on an RSpec produces one of three results:

1. an accept decision, meaning that the RSpec satisfies the policies configured into the policy engine;

2. a modified RSpec consistent with the configured policies, along with an accept decision; or

3. a reject decision.

This section provides an overview of how RSspecs, the two API calls, policies, and the Policy Engine fit together. An administrator configures a Policy Engine with two sets of policy rules: Outgoing and Incoming. The Outgoing rule set is applied on the RSpec output by LearnResources; these rules are invoked before the RSpec is returned to the caller, and can modify it to hide resource information that the caller should not see. The Incoming policy rules are applied to an incoming BindResources request before the RSpec is passed to the Aggregate Manager, which is responsible for actually binding the resources; these rules would typically operate on the Request portion or the RSpec to filter or modify incoming resource requests based on the caller’s identity. Section 4 contains examples of policy rules.

Figure 1 illustrates the process of a user discovering the available resources and binding a subset to Slice. First, the user calls LearnResources(Slice) on an aggregate to retrieve an RSpec consisting of the aggregate’s available capacity Cap and the resources currently bound to the slice Req. The Policy Engine intercepts the response returned by the Aggregate Manager and modifies Cap to hide some resources from the caller as specified in the Outgoing policy set, producing Cap’. The user then modifies the request portion of the RSpec to produce Req’, and submits it using BindResources(Slice, RSpec). The Policy Engine intercepts the incoming request and modifies Req’ to conform to policy set Incoming, resulting in Req”’. The Policy Engine could also choose to deny the request at this point; this is not shown in the figure. Finally, the approved request Req”’ is passed to the Aggregate Manager, which satisfies the request if the requested resources are available, and returns the status of the operation to the user.
When the Policy Engine processes a request, our model allows it to access information beyond the local policy rules and the RSpec. We refer to this additional data as the context of the request. The request context always contains the HRN of the user making the request and the HRN of the slice on whose behalf the request is made. The context can also include arbitrary fields from the aggregate’s local database of physical and logical resources. The Policy Engine is able to render its decision based on a rich set of information about the current state of the aggregate.

4 Examples

The Policy Engine is essentially a filter that accepts, rejects, or modifies resource requests based on a set of rules—functioning much like a firewall in front of the aggregate. We exploit this similarity in the design of sfatables, our policy engine implementation, which draws heavily on the Linux iptables firewall utility for inspiration. The next section will describe the sfatables design in detail. This section gives some example policies expressed using sfatables, with the intent of helping those familiar with iptables to intuitively understand the approach. It also gives example resource requests, again drawn from the set of network testbeds federating with PlanetLab.

4.1 iptables Analogy

Before giving specific policy examples, we describe the frontend interface that sfatables provides to resource owners, which very closely resembles the iptables tools. We chose this approach because specifying policies for resource-providing aggregates shares several characteristics with firewalls in IP networks. In particular, policies are defined relative to the request context, which may include the source address (corresponding to the HRN of the requesting party in our case). They are composed from a set of independent rules, each of which encapsulates a policy decision. Policies can be applied at many stages of request processing. For IP stacks, they may be applied for incoming or outgoing packets, before or after routing has taken place. For aggregates, they can apply when a user is trying to discover resources, when he is trying to request them, or trying to update them. For this reason, we look to a familiar tool—iptables—which is used for defining policies in IP networks. We base our Policy Engine framework, called sfatables, on its syntax and a set of abstractions that meet the aforementioned requirements: rules, matches, targets, chains, and hooks.

To help parse the other examples in this section, consider the following two sfatables rules:

```
sftables -A INCOMING --
  -m hrn --user-hrn-prefix plc.eu.inria.thierryp --
  -j ACCEPT
```

```
sftables -A INCOMING --
  -m hrn --user-hrn-prefix plc.eu --
  -j SLICE_COUNT --e node
  --max-nodes 150
```

Each rule has three segments separated by the -- token. A command (“-A” for add), which includes a chain (“INCOMING”); a match (“hrn”); and a target (“ACCEPT” or “SLICE_COUNT”). The first rule dictates that any request from user plc.eu.inria.thierryp be accepted. The second rule applies to users whose root provider is PlanetLab Europe. Such users are restricted to a maximum of 150 nodes per slice. Note the additional “-e” argument in the second rule. Each match and target is implemented separately for resource types known as elements, which are the building blocks of RSpecs. We currently support four elements: nodes, links, flows, and cores. What is important to note at this point is that the same target may have been used in the context of a different element, for instance limiting the number of links instead of nodes:

```
sftables -A INCOMING --
```
4.2 Example Policies

We now present several example policies, all of which except the last two are drawn from scenarios encountered on PlanetLab. The last two are hypothetical, but illustrate other scenarios in which sfatables might be applied.

4.2.1 Node Policies

We start with a policy for restricting foreign users’ access to the local aggregate. Often the main barrier preventing organizations from joining a federation is being able to distinguish between “foreign” and “local” users. For many networks, an easy way to start is to restrict users from a foreign aggregate to a small fraction of local resources. For example, the following rule may be used on the PLJ aggregate to limit users from PLE to a maximum of 20 nodes per slice.

```
sfatables -A INCOMING --m hrn --user-hrn-prefix plc.eu --
-j SLICE_COUNT -e node
--max-links 150
```

Next, we place a restriction on access to a subset of nodes, allowing them to be visible to only select slices. This is useful when nodes are contributed for the sole purpose of running a particular service (hosting specific slices) rather than general usage. In this example, the rule prevents these nodes from being advertised to other users by placing the rule in the OUTGOING chain.

```
sfatables -A OUTGOING --m hrn --user-hrn-prefix plc --
-j RESTRICT-DOMAIN -e node
--to-list tp.list
```

PlanetLab maintains a list of users that generate traffic that is susceptible to abuse complaints. This list is based on data from PlanetFlow, PlanetLab’s traffic auditing facility [26]. There is also a list of nodes that are especially sensitive to abuse complaints. The following rule prevents slices in the former category from running on nodes in the latter category.

```
sfatables -A INCOMING --m slice-tag --slice-tag scanner --
-j RESTRICT-DOMAIN -e node
--drop-list sensitive-nodes.list
```

A somewhat more complex rule involves giving a slice 1/10th of the number of public IP addresses as the number allocated by its site.

```
sfatables -A INCOMING --m user --negate --user-role admin --
-j RESTRICT-SITE-NODE-COUNT -e node
--max-nodes 1000
```

4.2.2 Link Policies

In networks that support physical topologies, administrators may wish to place restrictions on link bandwidth, rather than on nodes. For instance, the following rule limits a slice’s per-node bandwidth to 1 Mbps if it is in a slice whitelist, and rejects the request otherwise.

```
sfatables -A INCOMING --m slice-whitelist
--whitelist vini.list --
-j RESTRICT-PROP -e link
--max-bandwidth 1024
```

```
sfatables -A INCOMING --m all --
-j REJECT
```

4.2.3 Flow Policies

We also use sfatables to mediate access to an aggregate of OpenFlow switches [21], where slices are allowed to request flow specs that they control. The first prevents users from acquiring the complete flow space of a switch.

```
sfatables -A INCOMING --m pattern -e flow
--flow-spec */**/**/**/**/**/**/ --
-j REJECT
```

The second drops requests involving DHCP, with the protocol set to 17 (UDP), the source or destination port set to 67 or 68, and the remaining fields set to ‘.’ or “don’t care.”
4.2.4 Excess Resources Policies

Suppose Amazon’s EC2 wanted to contribute a fraction of its unused computing resources to a research testbed such as GENI. The following sequence of rules might apply. The first ensures that the facility is accessible to participating universities.

sfatables -A INCOMING --
-m pattern -e flow
--flow-spec
_/_/17/67-68/_/67-68/_/67-68/_/_/--
-j REJECT

A second rule ensures that a single slice is limited to a maximum of 30 cores.

sfatables -A INCOMING --
-m site --negate
--whitelist ok-sites.list --
-j REJECT

A third rule removes a set of hot nodes.

sfatables -A INCOMING --
-m all --
-j RESTRICT-DOMAIN -e node
--drop-list hotnodes.list

A fourth rule limits the total number of cores from the research network to 20% of the remaining capacity.

sfatables -A INCOMING --
-m all --
-j LIMIT-SITE-COUNT -e core
--limit-to-buffer 0.2

4.2.5 Pricing Model

The following example implements a hypothetical pricing model, whereby the first quantum of resources allocated is free, but charges start to apply when it exceeds a certain threshold.

sfatables -A INCOMING --
-m core --min 5 --
-j CHARGE-EXCESS -e core
--threshold 5 --rate 2

4.3 Example Requests

The following illustrate the kinds of requests users make in the federated PlanetLab. Rather than present raw XML, we present them in a “sugared” form that is easier to read. This sugared notation is in fact supported by PlanetLab’s request-allocation tool chain through the use of XSugar [7], a mechanism for defining an equivalent but more readable format for XML data.

4.3.1 PlanetLab

On PlanetLab, a user can bind his slice to a set of nodes by enlisting them in his request RSpec. In addition, he can also reserve certain resources on these nodes. The following RSpec requests three nodes at Princeton along with a bandwidth limit and a disk quota on each node.

Request Princeton_nodes

Node "planetlab-01.cs.princeton.edu"
BwLimit 1000MB
DiskLimit 10GB
Node "planetlab-02.cs.princeton.edu"
BwLimit 8000MB
DiskLimit 10GB
Node "planetlab-02.cs.princeton.edu"
BwLimit 900MB
DiskLimit 10GB

4.3.2 VINI

A VINI RSpec consists of a collection of virtual links. The following request is for a virtual topology mirroring the Internet2 physical topology. It allocates a sliver to the slice on one node at each VINI Internet2 site (Atlanta, Chicago, Houston, Kansas City, Los Angeles, New York, Salt Lake City, Seattle, and Washington, D.C.) and connects slivers on nodes that are adjacent in the Internet2 topology by 1Mb/s virtual links. For example, the sliver in Atlanta (on node i2atlal1) is connected by virtual links to slivers in Chicago, Houston, and Washington, D.C.

Request Internet2_topology

i2atlal1 -- i2chic1 [1Mbit]
i2atlal1 -- i2hous1 [1Mbit]
i2atlal1 -- i2wash1 [1Mbit]
i2chic1 -- i2kans1 [1Mbit]
i2chic1 -- i2wash1 [1Mbit]
i2hous1 -- i2kans1 [1Mbit]
i2hous1 -- i2losa1 [1Mbit]
i2kans1 -- i2salt1 [1Mbit]
i2losa1 -- i2salt1 [1Mbit]
i2losa1 -- i2seat1 [1Mbit]
i2newy1 -- i2wash1 [1Mbit]
i2salt1 -- i2seat1 [1Mbit]
4.3.3 OpenFlow

PlanetLab has recently begun a pilot federation arrangement with Stanford University’s OpenFlow deployment. Through this peering arrangement, certain PlanetLab users will be able to manipulate traffic routes involving Stanford’s PlanetLab nodes. A user can request that a certain flow be directed to a PlanetLab node using an OpenFlow RSpec. The following RSpec sets up a link between two PlanetLab nodes connected to the same OpenFlow switch. This link carries all packets tagged with the VLAN 2004.

```
SwitchEntry
  Node
    Id 3
  InterfaceEntry
    Port 5
    RemoteNodeId 3
    RemotePort 6
  FlowEntry
    Policy readonly
    Vlan_id 2004
```

5 Design and Implementation

The full sfatables tool chain consists of the six components shown in Figure 2. Briefly,

- Administrators use the frontend to add sfatables policy rules using a simple but powerful policy specification language. Our policy language is based on an iptables-like syntax, as illustrated by the examples in Section 4. The frontend also provides an interface for modifying, deleting, and listing sets of policy rules.
- The compiler transforms a given set of policy rules into a policy program. The policy program is at the heart of sfatables. Policy programs are composed in XSLT, a standard language for transforming XML documents.
- The XPath query interface provides access to both global and local state. XPath is a query language for XML documents. It is used to create a request context for each incoming request, containing request-specific variables such as the user and slice HRNs associated with the request.
- The data abstraction layer translates XPath expressions to queries on the SFA global registry and the aggregate’s local resource database. New types of aggregates can be plugged into the sfatables framework by modifying the data abstraction layer to reflect the aggregate’s database schema.
- The runtime uses the data abstraction layer to create a request context for each call to LearnResources or BindResources. It then links the request context to the appropriate policy program, and computes on the RSpec associated with the call. The output of the runtime is a decision (accept or reject) and a (possibly modified) RSpec.
- A query engine can be used by a user or peer to query a policy to ask questions such as “what is the maximum number of nodes that I can allocate?” This may help the user to craft an RSpec that is consistent with an organization’s policies.

The frontend was described and illustrated in Section 4. The query engine is described in Section 6. The rest of this section describes the other components.

5.1 Compiler

Our goal is to specify policies in a way that they possess three qualities. First, they should be unambiguous in the meaning they convey, in a way that representatives of peering organization can use them to formally establish peering agreements. Second, they should be in a format that makes them amenable to analysis by policy-makers and administrators. Policy makers should be able to “query” policies to extract the desired information without having to master the specification language or reason about the content of a document at length. Third, they
should be portable so that they can be interpreted by users on commodity platforms. Policies are not only important from the perspective of administrators and policy-makers, but are also critical from the perspective of users. Users need to be informed about policies so that they can allocate requests wisely by virtue of being able to unambiguously determine what they can and cannot allocate.

The design of our policy framework revolves around these requirements. An important design decision in our framework was to use the XSLT language for representing policies. XSLT is portable—it is a standard that is widely supported on commodity platforms—and can even be loaded in web-browsers. It uses the declarative programming paradigm, which makes it convenient to analyze and query (more on this in Section 6). It is Turing-complete, allowing it to express arbitrarily complex policies. Additionally, it is dedicated to processing XML, which is the notation used to define RSeps, making it a natural tool to use in our work. Both matches and targets are implemented as XSLT programs: a match inspects a request context and an RSpec and determines if it matches a set of criteria, and a target is a filter that edits an RSpec to apply a set of restrictions to it.

The policy compiler takes as input a set of rules and compiles them into an XSLT policy program, which we call the \textit{global policy}. Since the rules that make a policy are independent, the process of compilation simply combines the following rule functions into a pipeline in XSLT. Additionally, it adds the administrator-provided context for the match or target (e.g., “max-links” in earlier examples) as an argument to the match; we’ll discuss contexts more in Section 5.2.

\begin{verbatim}
function main =
  recurse_list($whitelist)

  The \textit{sfatables} tool converts the command-line parameters passed to it into XML, e.g., \texttt{--user-hrn ple.inria.thierry@ple.inria.thierry} is converted to the following fragment and plugged into the match.

  <argument>
    <name> user-hrn </name>
    <help> HRN of the requesting user </help>
    <operand> HRN </operand>
    <value> ple.inria.thierry </value>
  </argument>

5.2 Data Contexts

A context represents a collection of data values that can be read inside an XSLT match or target. The values in each context are organized as an XML document tree, and can be retrieved using XPath expressions. XPath is a domain-specific language that computes a subset of items in an XML document that match a given set of criteria. Syntax-tactically, XPath expressions are like file paths in UNIX, with some additional features. In the whitelisting example above, there were two XPath expressions:

\begin{verbatim}
let whitelist=//admin-ctx/whitelisted-users/hrn
let current_user_hrn=//request-ctx/user/hrn

sfatables currently provides two contexts to XSLT matches and targets: an \textit{admin-ctx} containing administrative information, and a \textit{request-ctx} containing values relevant to the specific request. In the above example, the list of whitelisted user HRNs from the admin-ctx is retrieved the first XPath expression, and the second returns the HRN of the requesting user. We will revisit XPath in more detail in Section 6.2.2.

Values within the two contexts can come from the global SFA registry (which can be queried to learn about named principles), the aggregate’s local resource database, or local configuration files. A data abstraction layer is used to create the contexts by building XML nodes using the results of arbitrary data access operations (e.g., database queries, API calls, or shell scripts). Administrators can add new types of data to the contexts by modifying this layer.

5.3 Runtime

sfatables currently intervenes at two points in request processing: exposing resources via \textit{LearnResources}, and allocating resources via \textit{BindResources}. The abstraction used to differentiate rules that apply to one point from the other is called a 'chain', as in iptables. The ‘OUTGOING’ chain, which applies to the output of \textit{LearnResources}, can be used to hide certain resources
from slices. Similarly, the 'INCOMING' chain, which applies to the input of BindResources is used to deny or restrict access to resources that are known to exist.

To execute rules in a given chain, the sfatables runtime first constructs the request context. The request context required by each match and target is declared as part of its specification using XPath expressions. The database that these expressions refer to contains some compulsory fields, such as the slice and user HRNs, supplied by sfatables. It also contains items specific to the aggregate, and provided by the aggregate. For example, the PlanetLab aggregate contains several policies that operate on properties of a user’s site, such as the amount of resources that the slice contributes. Information about a site, the nodes provided by it, and so on, is therefore included in the aggregate-specific context.

Once the request policy has been computed,

\[
\text{request-policy} := \text{global-policy (request-ctx)}
\]

it is applied to the incoming or outgoing Rspec:

\[
\text{output-rspec} := \text{request-policy (input-rspec)}
\]

6 Policy Analysis

It is crucial for users and policy-makers alike to understand what a given policy entails. For policies involving a handful of rules, what is allocatable can often be easily visualized. For complex policies involving tens of rules understanding the meaning of a policy can be problematic. This is especially true when the constituent rules impose numerical limits on requests, limiting the total number of nodes, the end-to-end bandwidth between certain nodes, and so on, rather than simply blocking a subset of resources.

This section considers the analysis of resource-allocation policies that have already been compiled into XSLT. There are two ways of inspecting these policies at a meta level: (1) by executing the corresponding policy program, and (2) by querying it programmatically by first transforming it into a logic program. We begin by describing some scenarios in which analysis is critical, then we describe the mechanisms that make analysis possible.

6.1 Scenarios

We present three examples of operations on policies. The first selects a given request out of a pool of candidate requests. The second validates one peer’s policy against the requirements of the other. The third involves the composition of policies across multiple aggregates to yield a composite policy.

<table>
<thead>
<tr>
<th>Site</th>
<th>Nodes</th>
<th>Bandwidth (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 3: An example of a querying a real policy.

Policy-driven Allocation. Users can analyze policies in two ways. If, as is often the case, the space of possibilities is small, then a user can “execute” the policy for each candidate request using standard implementations of XSLT. He could do so by loading the policy in a web browser, or by writing a Python script that executes the policy using the libxslt library. This direct execution of policies gives users a convenient and rapid method of selecting valid requests out of a collection of requests that meet their requirements.

For example, consider a case in which a user would like to route packets through a series of m geographically-local clusters of node. Accordingly, the user can write a script that generates candidate routes by picking one node each from the m clusters for every route, then apply the prevailing policy to each candidate route until he finds one that works.

Policy Validation. When the space of possible requests given an allocation policy is large, it is helpful to approach the search of the allocation space using logic programming. In such cases, we support the mechanism of analyzing policies using Prolog by transforming the XSLT policy into a Prolog knowledge base.

Consider a peering agreement whereby peer A has promised to provide a minimum resource allocation to B. This allocation allows users of B to get 5 nodes, to reserve 25Mbps of bandwidth across these nodes, and to do this with the additional restriction that not more than 3 nodes are allocated out of either of two dominant geographically-local node clusters.

Peer A has a policy of limiting allocations on a per-site basis. Across the four sites in the two clusters, it enforces the node and bandwidth limits in Figure 3.

To verify that the policy supports the minimal allocation, an administrator can enter the following query:

\[
\text{compute_node_count (Rspec \_W)}, \quad \text{compute_total_bw (Rspec \_X)}, \quad \text{site_count (Rspec \_[1,2]\_Y)}, \quad \text{site_count (Rspec \_[3,4]\_Z)}, \quad \text{W\#>=5, X\#>=25, Y\#<=3, Z\#<3}.
\]

When evaluated with the rules of the policy along with the contextual information required for it to operate, the query reveals a feasible result involving the allocation of
two nodes at Site 2, one node at Site 3, and one at Site 4. Recall from Section 5 that sfatables inserts the contextual information needed for a policy to operate into the policy document. In this case, this information includes the per-site limits and the mappings between nodes and sites.

**Policy Composition.** We are working to extend SFA’s resource-allocation framework to support policy composition. Although we have not yet incorporated it into our model, we mention it in this section for completeness, as it is one of the goals of our design.

It may sometimes be desirable to take the policies of two aggregates and compose them. The SFA federation framework defines an element called an “Aggregate of Aggregates” (AoA) that arbitrates a user’s access to several aggregates. Via an AoA, a user can access the resources hosted by all of the aggregates through a single interface. For example, a single AoA arbitrates access to PLC, PLE, PLJ, and VINI, where a given user might try to acquire resources across the entire set. In effect, the policies at the AoA can be an arbitrary composition of the policies of the individual aggregates. An AoA can simply concatenate the effect of these policies, or it can extend it by inserting its own policies.

Since XSLT-based policies are executable, such composition is greatly simplified. In fact, policies are composed to create aggregate-policies in the same way as rules are concatenated by the sfatables compiler to create policies. The policies can then be applied to a request sequentially, and the output of one policy can be used as input to the next.

### 6.2 Transforming Policies into Prolog

This section outlines the mechanisms for transforming XSLT policy documents into Prolog. Our goal is not to describe this transformation in a way that it can be reproduced, but rather to build up enough context for the reader to appreciate the flexibility of our policy engine. For the purpose of this explanation, we use a model in which policy documents take as input an RSpec in XML format, and give as output a True or False value depending on whether or not the request is acceptable. The transformation itself consists of two steps: transforming the data to be processed, and transforming the policy.

#### 6.2.1 **RSpec Documents**

We transform RSpec documents and all intermediate data in XML into ordered trees in Prolog. An ordered tree is defined using three terms: an identifier, a set of attributes, and a sequence of ordered subtrees. Attributes and children are represented using lists of values. Since both XSLT and Prolog are dynamically-typed languages, we are not required to carry out the tedious task of implementing a type-inference algorithm for XSLT as a preliminary step.

Figure 4 illustrates an RSpec in standard XML format, and its Prolog counterpart. As seen in the example, all values in the Prolog code are quoted as strings. This decision is explained in Section 6.2.2.

#### 6.2.2 **Policies**

The declarative nature of XSLT programs enables us to transform policies through a syntax-directed transformation on the source AST, without the need for any accompanying data- or control-flow analyses, as it would have been necessary to use for a programming language that included support for imperative constructs. The XSLT language can be divided into two parts: the XPath query language, which computes arbitrary subsets of nodes in an XML document; and a functional language for operating on such subsets. XSLT also has a template-matching abstraction used for formatting documents. We ignore this abstraction because it is mainly geared at annotating XML documents for display, and is not especially useful for defining policies. In addition, we assume that the program, input, and output are well-typed.

XPath queries translate naturally into Prolog. To demonstrate how, we first present a brief overview of XPath. As mentioned earlier, XPath expressions look like UNIX file paths. As an example, in the document in Figure 4, the XPath:

```
/rspec/nodespec/node[@fqn="planet1.cs.mit.edu"]/core
```

refers to both node elements. Additional selection criteria can be added as predicates to the path:

```
/rspec/nodespec/node[@fqn="planet1.cs.mit.edu"]/core
```

The above query returns only those cores that are under the specified node. Next, XPath provides several relations (called axes) in addition to the child relationship implicit in the ’/’ used to punctuate file paths, including descendant, parent, sibling, following, preceding, ancestor, and so on. For instance, one way of selecting all available cores in the RSpec in Figure 4 is to use the following expression:

```
/rspec/ descendant::node[child::core]
```

It is tempting to translate XPath expressions into constraints on node sets. Accordingly, the expression

```
Ctx/child:E is transformed into
childtree(T,S):=&lt; T n o d e ( U , V , S x ) &gt; , member(S,Sx).
child(Ctx,E,X):= childtree(Ctx,X), X n o d e ( E , J , J ) .
```

However, since XPath and XSLT frequently treat the output of such expressions as lists, we would need to resort to the use of extra-logical predicates in Prolog such as findall and bagall to collect these solutions. Such
predicates are problematic, and at best extremely inefficient to re-execute with backtracking. Our goal is not to apply policies to requests, but to discover feasible requests given a policy and a set of requirements. With this in mind, we take special care to maintain the property of backtrack-ability in the generated code. All XPath operations, and operations on nodes are recursively applied to lists as sublist constraints. The definition of the child axis using this approach is given below.

\[
\text{child} \text{rec}(L, E, I, Result) := \\
\text{Result} = I. \\
\text{child} \text{rec}(ChildList, E, I, Result) := \\
\text{A} = \text{o} \text{tree}(E, \text{Attrs, CA}), \\
\text{Newl} = [A] \text{[I]}, \text{append}([A, Newl]), \text{child} \text{rec}(D, E, Newl, Result). \\
\text{parent} \text{rec}(Ts, E, I, Result) := \\
\text{Ts} = [], I = \text{Result}. \\
\text{parent} \text{rec}(Ts, E, I, Result) := \\
\text{Ts} = [A] \text{[D]}, \text{A} = \text{o} \text{tree}(E, \text{Name, Attrs, Children}), \text{child} \text{rec}(\text{Children}, E, [], \text{CA}), \text{append}([\text{CA}, \text{Newl}]), \text{parent} \text{rec}(D, E, Newl, Result). \\
\text{child}(Ts, E, Result) := \\
\text{parent} \text{rec}(Ts, E, [], \text{LocalResult}), \text{sublist} \text{(Result, LocalResult)}. \\
\]

It is straightforward to imitate this function to support other axes such as descendant, attribute, sibling, and so on. A complete XPath query in Prolog starts with a context node followed by a series of steps, where each step is coupled with a predicate where appropriate. For example, in the previous example, the context node is set to the root node, rspec. Next, we move one step along the descendant axis, and another step along the child axis. Accordingly, the previous example translates into this equivalent in Prolog:

\[
\text{root}(\text{rspec, X}), \text{descendant}(X, \text{node, Y}), \text{attribute}(Y, \text{fqn, ['planetlab1.cs.mit.edu']}), \text{child}(Y, \text{core, Z})
\]

Since the bulk of operations in XSLT are implemented using XPath, transforming XPath expressions is a significant step forward in the transformation of policies. We have started with a subset of the XSLT language that excludes control structures such as for-each, instead using recursive function calls to implement the same behavior in sfatables policies. Note that under the assumption that the policy returns a True/False value for incoming requests, these control structures follow declarative semantics, which simplifies the transformation problem.

We conclude this outline with a note about the translation of data values, and the reason that all values are rep-

Figure 4: The XML and Prolog representations of an RSpec, side-by-side.
represented as strings on the Prolog side. First, we run into a syntactical problem when translating values from XML to Prolog – all XML values have the same representation (double-quoted string), whereas Prolog syntactically differentiates between symbols, numbers and strings. Since one of our objectives is to circumvent the problem of type inference, we add a runtime check to address this problem. Specifically, we translate all XML values into Prolog strings, and post-process the resulting program to preprend a type predicate (number_atom/2) to each predicate that expects a numerical operand.

7 Related Work

This work addresses one modest piece of the larger problem of federating resources contributed by multiple autonomous organizations. This section discusses the larger design space, and in doing so, comments on related work.

At a high level, the idea of federating access to resources across multiple domains is being pursued in many settings, from the Grid [19] to global network testbeds [13] to clouds [2, 14, 9, 28]. In each case, the value to users is to gain seamless access to a larger collection of resources than is available in any one domain, and the value to resource owners is to make those resources available (perhaps in return for some compensation) under the control of a single framework rather than having to deal with each user on a case-by-case basis.

Within that context, there are several, mostly orthogonal issues. The first is a common set of interfaces for accessing and controlling those resources. Such interfaces can emerge form non-federated systems (e.g., EC2 [2] and GoGrid [14]), or they can be designed explicitly to support interoperability across multiple independent systems (e.g., DeltaCloud [9] and Reservoir [28]). We adopt the SFA interfaces that were influenced by PlanetLab, but generalized for the full diversity of testbeds expected to be included in GENI. While we make no claim that this particular interface is either fully general or complete, we do note that operations to learn the available resources and request an allocation of those resources are universal across the federated systems outlined above.

A second issue is federated identity management (FIM). Again, we use the SFA, which requires the caller pass the appropriate credentials to each operation. Such explicit delegation of trust is certainly not novel [6, 15, 31, 8], but our approach to policy does place one requirement on the identity management system—it must be possible to anchor the policy rules with a human-readable name that represents the underlying chain of trust. Being able to specify a rule in terms of the name plc.princeton.coblitz—implying that the owner trusts plc which trusts princeton which in turn trusts coblitz—is important from a usability point-of-view. It also makes it possible to treat groups of slices in a similar way, for example, defining a rule that applies to all slices indirectly trusted by plc.eu.

The most notable FIM in use today is SAML [29] / Shibboleth [30], which supports single sign on (SSO). The idea is to allow a user of one organization to use credentials provided by that home organization to identify him or herself at other, federated organizations. (In general, a user acquires assertions from an identity provider and presents them to a service provider, who uses them to make an access control decision.) In contrast, the SFA uses a more general PKI-like approach to establishing identity, but this is largely orthogonal to sfatables—the only relevant issue being whether the FIM permits rules that correspond to classes of principals (e.g., plc.eu or if it forces administrators to define a set of pairwise rules (i.e., one for each principal); it’s not clear how SAML supports such rules. However, the important issue is how one defines resource allocation policy given proof of the principal’s identity. In principle is seems likely that any FIM (including SAML) could use many of the mechanisms we’ve built into sfatables, but actually demonstrating how they would be plugged in remains future work.

This gets us to the third issue, which is the focus of this paper: How resource owners express, analyze, and enforce resource allocation policies. We make two points on this topic. First, an important property of our approach is that it distinguishes between the “security” policy underlying identity, and the “allocation” policy managed by sfatables, with each potentially managed by separate machinery. For example, it would be possible to define and manage a security policy using KeyNote [5] or SD3 [16], and then define/manage the resource allocation policy using sfatables. This decoupling is by design: having a credential gives the caller the right to ask for resources, but the callee gets to decide on a case-by-case basis what resources it is willing to grant. Allowing for two distinct mechanisms—rather than have a single mechanism that spans the entire policy space—is rooted in a philosophy that the underlying security architecture is system-wide, while deciding how to allocate local resources is a purely local decision. We do not attempt to embed resource allocation assertions in credentials.

Second, declarative languages are a natural way to express allocation policies. Certainly SD3 does this, as it is an extension of Datalog. Our approach uses XSLT—and related XML tools—largely for their universality. As shown in Section 6, however, it is possible to translate between XSLT and Prolog, for example. On the flip side,
our work is well-aligned with the idea of users specifying constraint-based resource requests, which can be used as a basis for building self-managing services [32, 33]. We believe this approach will increase in relevance with the increasing complexity of resource-providing substrates, as they provide a natural paradigm for thinking about resources in such settings. Deriving constraints from sfatables policy documents complements such efforts, since the prevailing allocation policy is an important factor to consider when optimizing a resource allocation. In turn, we would also like to incorporate the idea of using a cost function in our own framework and to consider extending the current SFA client tools to support constraint-based requests.

A fourth related issue is how a system moderates users' appetite for resources. This sometimes motivates a market for buying and selling resources, in which rights to resources are managed by third-party brokers [11, 3, 17]. Our approach is different from a market-based approach in the sense that it gives owners the ability to directly say how many resources a given user (or class of users) may acquire, whereas market-based approaches let the market decide. Whether the two approaches are competitive or complementary is a matter of perspective. On the one hand, a policy-based system like sfatables can be given rules to grant one subset of resources to specific users and another subset of resources to a broker for redistribution. In this sense they are complementary. On the other hand, the designers of market-based systems argue that policies can be supported “on top of” markets by manipulating the market in certain ways. Our experience on PlanetLab is that resource owners wish to retain very specific control over who gets what resources, with market-based allocation systems most useful for allocating “extra” resources not otherwise spoken for by any policy rules.

8 Conclusion and Future Work

We have presented a framework for policing resource allocation in a federated setting that caters to the entire spectrum of participants including users, administrators, policy makers, and developers.

• Users benefit from a uniform policy-definition and resource-allocation framework that cuts across organizations and resource types. They interact with these testbeds using a familiar interface, specifying resource-allocation requests in the notation of RSspecs.

• Administrators can deploy allocation policies using a familiar firewall-like interface. Like in the case of firewalls, the policy engine runs independently of the complex security machinery of aggregates, simplifying the process of defining rules.

• Policy makers benefit from the unambiguous nature of the policy. The “firewall” rules specified by administrators are compiled into a policy program in a standard language (XSLT), which can be executed for a given request in order to evaluate if the request is acceptable under that policy or not. We have described an approach involving the use of Logic Programming to gain a deeper understanding of policies.

• Our policy-definition framework is extensible. New resources can be added in the form of “matches” and “targets” specified using XML and XSLT. This enables developers to add support for new resource types in a platform- and language- independent way.

We have already released a first version of sfatables, available in the PlanetLab SVN repository [27]. We hope that as it is progressively deployed across the growing span of the PlanetLab federation, administrators will continue to contribute new resource types and new policies. We are especially interested in coordinating with frameworks such as Rhizoma [33], which automate the process of resource allocation, so that they incorporate the prevailing allocation policy within this allocation.

References


