sfatables: A firewall-like policy engine for federated systems

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

Abstract—Recent efforts to federate computation and communication resources across organizational boundaries face a challenge in establishing the policies by which one organization’s users can access resources in other organizations. This paper describes an approach to defining, communicating, analyzing, and enforcing resource allocation policies in this new setting. Our approach was designed to address the needs of PlanetLab, but we demonstrate through a range of examples that it is general enough to accommodate a diverse collection of computing facilities. Our policy engine is implemented in a specific tool chain, called sfatables, that is patterned after the iptables mechanism used to define packet processing policies for network traffic. The interface to our policy engine thus uses the familiar paradigm of a firewall and provides a flexible interface for resource owners to specify access policies for their resources. Our implementation makes it possible to precisely document policies, query, and analyze them.

I. INTRODUCTION

Computational resources are becoming increasingly commoditized. With a growing demand among application programmers, service developers, and researchers to acquire compute, storage, and network resources from multiple sources (e.g., commercial and academic clouds, programmable network testbeds), there is increasing interest in federating resources from multiple autonomous organizations into a seamless and ubiquitous resource pool [11], [19], [18], [2], [12], [7], [25]. Federation benefits resource consumers by providing a common interface for accessing the widely distributed and diverse collection of resources they need. Federation benefits resource providers by extending their reach to a wider audience as well as defining a single coherent mechanism for controlling who is allowed to access what resources.

Federating compute, storage, and network resources across organizational boundaries is an explicit goal of PlanetLab [23], GENI [11], and Open Cirrus [19]. Federation is the mechanism that connects PlanetLab to several other facilities, such as Measurement-Lab (M-Lab) [16], an open platform for researchers to deploy Internet measurement tools, and VINI [4], a layer-2 variant of PlanetLab deployed on the Internet2 and NLR backbones. Our experience with federation in PlanetLab has shown that, before resource owners are willing to contribute their resources to a global facility, they must be able to control how users can access those resources. For example, Measurement-Lab (M-Lab) [16] and EmanciLab [8] wish to limit their facilities to community-selected experiments that are consistent with the system’s purpose: supporting broadband measurement experiments and network management experiments, respectively. Testbeds like VINI want admission control so as to not overload their resources. In another case, G-Lab [10] (a VINI-like testbed deployed on the German research network) is restricted by funding sources to ensure that all German researchers are given resources before any international researchers can have access. A similar situation exists in Open Cirrus, where corporate policies and business relationships dictate what remote users are allowed to access each cluster.

In sum, the problem of specifying resource allocation policies is a barrier-to-entry for joining the PlanetLab federation. From our experience we can identify two high-level requirements that a resource allocation policy framework must satisfy:

- **Resource providers** must be able to express and enforce a wide range of policies across a rich set of physical and logical resources.
- **Resource consumers** must be able to discover the policies that apply to them, so that they can take these policies into account when requesting resources.

Two design decisions underlie our approach to building such a framework for PlanetLab. 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 iptables—a popular 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 implement policies free of side effects. This decision makes policies easier to read and greatly simplifies analysis and composition. To this end, policies are defined as filters that operate on resource specification documents called RSpecs. RSpecs are used by resource providers to publish available resources and by federation users to request resources. Accordingly, a policy transforms an input RSpec into a new RSpec that satisfies a given set of constraints. In this way, all request state is isolated in the RSpec data structure, thereby simplifying the operating interface of a policy. Since we specify RSpecs using XML notation, this interface is a simple XML transformation.

In our implementation, the sfatables tool transforms policy rules specified in an iptables-like syntax into XSLT programs. XSLT, a declarative language for transforming XML documents, seems a natural choice for our purpose. However, using 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 resource providers and consumers.

This paper makes three contributions. First, it proposes a general model for establishing resource usage policies in federated systems. To the best of our knowledge, sfatables is the first implementation of a solution to this problem at the scale and diversity of the PlanetLab federation. This model is described in Section II. Second, it enumerates the collection of policies that we have developed and that have proven to be useful in PlanetLab. We posit them as examples of policies any federated system will need to support. These policies are discussed in Section III. 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 documenting an organization’s policy. The implementation and analysis tools are presented in Sections IV and V, respectively.

II. MODEL

Federating resources contributed by autonomous organizations is a multi-faceted problem. It typically implies a common set of interfaces [2], [12], [7], [25], a shared identity management system, 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.

To focus on this topic in a concrete way, we describe our approach to establishing resource allocation policies in the context of a specific federated system: PlanetLab. This section presents enough background about PlanetLab’s approach to federation to understand the context for sfatables, but its primary contribution is to describe a model for how resource allocation might work in any federated system.

A. PlanetLab Deployment

The “public” PlanetLab is a global testbed for network and distributed systems research, consisting of approximately 1000 machines spanning nearly 500 sites in over 40 countries. Research institutions contribute machines and connect them to their campus networks, and in return, researchers at those institutions are granted access to the global pool of machines. Each PlanetLab node runs a common software stack that implements a slice abstraction—a network-wide resource container that hosts user experiments.

Originally there was only one PlanetLab, with a centralized control site—known as PlanetLab Central (PLC)—responsible for operating the global platform. However, the public release of the PlanetLab software, allowing any organization to bring up their own “private” PlanetLab, has had two results. First, the public PlanetLab is now a decentralized federation of regional testbeds maintained and operated by independent organizations such as PlanetLab Europe (PLE) and PlanetLab Japan (PLJ), in addition to PLC. Each runs its own instance of the PlanetLab control software and maintains and operates its own nodes. Second, about a dozen private PlanetLabs of non-trivial size have been created around the world by governments, universities, and corporations. Many of these facilities have joined into the PlanetLab federation with the condition that they can maintain control over their local resources. These facilities include Measurement Lab (M-lab) [16], G-Lab [2], ProtoGENI and Open Cirrus. The question this work addresses is how to give PlanetLab’s existing
federation partners more autonomy, and attract new partners into the federation, by providing fine-grained control over how resources are allocated to federation users.

B. Federation Architecture

PlanetLab supports federation through a set of interfaces called the Slice Federation Architecture (SFA) [24]. 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 on resources:

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

both of which are exported by an aggregate—the abstraction representing a collection of computation and communication resources on behalf of some organization. The regional testbeds such as PLE and PLJ are aggregates, as are private PlanetLab networks such as M-Lab, VINI, EmanciLab, and G-Lab. We refer to these functions as the SFA Resources API. Each aggregate runs a server called the Aggregate Manager that implements this API.

The Slice argument to the SFA Resources API is a cryptographically signed certificate representing the principal invoking the operation. For the purpose of our discussion, we focus on one field in that certificate—a human-readable name (HRN) for the slice. For example, "princeton.colblitz" is the HRN for a slice running the CoBlitz network service, "princeton.colblitz" is the HRN for a slice running the Network Diagnostic Tool (NDT), and "princeton.vivaldi" is the HRN for a slice experimenting with the Vivaldi Internet coordinate system. Each HRN implies a chain of authorities that take responsibility for the slice—corresponding to a certificate chain—with the PlanetLab Consortium (PLC) at the root. We focus on the HRN for a slice because it anchors the policies an aggregate owner might wish to define. For example, M-Lab might be willing to allocate a slice across all of its servers to "princeton.colblitz" but not to "princeton.vivaldi". Moreover, aggregate owners could take advantage of the structure of an HRN to define policies that apply to, for example, all Princeton slices ("princeton.*") or all PlanetLab slices ("*.colblitz").

C. Resource Specifications

The RSpec argument of the SFA Resources API contains a resource description encoded as an XML document. The RSpecs in use in the PlanetLab federation contain a Capacity statement that describes the available resources, and a Request statement indicating the resources currently allocated to Slice (or a sample request if Slice has no resources). In the SFA’s resource allocation model, the LearnResources call returns an RSpec describing the resources available to be allocated and those already allocated to the Slice, while the BindResources call binds the resources specified in the Request to the names Slice. Between calling the first and second operation, the user has an opportunity to engage other mechanisms (e.g., resource discovery services [1], [21], [22]) to help transform the Request statement returned by LearnResources to reflect the desired resource allocation. Each BindResources call for Slice supersedes all previous calls; the resources bound to Slice are changed to match the submitted RSpec.

The SFA permits each aggregate to define its own RSpec format. The following illustrate the kinds of resource 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 supported by PlanetLab’s request-allocatin tool chain through the use of XSugar [6], a mechanism for defining an equivalent but more readable format for XML data.

On PlanetLab, a user can bind his slice to a set of nodes by listing them in the Request statement in the 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 5000MB DiskLimit 10GB
  Node "planetlab-03.cs.princeton.edu" BwLimit 5000MB DiskLimit 10GB
```

VINI extends the PlanetLab RSpec by adding virtual links between the nodes. Virtual links are identified by their endpoints and allocated bandwidth. The following request is for a virtual topology connecting an Internet2 node in Atlanta to nodes in Chicago, Houston, and Charlotte, N.C., by 1Mbps virtual links.

```
Request Atla_topology
  Node "node.atla.internet2" DiskLimit 5GB
  Node "node.hou.internet2" DiskLimit 5GB
  Node "node.chi.internet2" DiskLimit 5GB
  Link Endpoint "node.atla.internet2" Endpoint "node.chi.internet2" Bitsps 1000
```
PlanetLab has recently begun a pilot federation arrangement with Stanford University's OpenFlow deployment. OpenFlow provides an interface for programming the flow tables of network switches. 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

D. Resource Allocation

Our current work extends the SFA Resources API to incorporate a Policy Engine so that aggregate owners can specify access and allocation policies for their resources. The Policy Engine runs as part of the Aggregate Manager, and operates on the RSpec arguments of the LearnResources and BindResources calls. Running the 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 deny decision.

This section explains how RSpecs, the two Resource 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 III contains examples of policy rules.

Figure 1 illustrates 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 but this is not shown. Finally, the approved request Req” is passed to the Aggregate Manager, which grants the request if the resources are available, and returns a status code to the user.

Our model allows the Policy Engine 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 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.

III. POLICIES

The Policy Engine is essentially a filter that accepts, rejects, or modifies resource requests based on a set of rules, independent of the aggregate—functioning much like a firewall.

The next section describes the sfatables design in detail. This section gives an insider's look at policies that we have implemented on federated systems that we manage, as well as how they are expressed using sfatables.

A. sfatables Interface

Before giving specific policy examples, we describe the frontend interface that sfatables provides to resource owners. sfatables borrows from iptables the abstractions of rules, chains, matches, and targets. Rules are independent policy elements. Since policies can be applied at multiple points in request processing (i.e., when the user is discovering, requesting, or updating resources), chains group together rules depending on the stage of processing at which they apply. For example, all rules used to hide resources from users calling LearnResources are grouped on the same chain.

Each rule is composed of a match and a target. Matches are predicates that classify requests, and targets define actions to apply in the event of a match. One action that a target can take is to modify the RSPEC. Matches and targets can access information inside the RSPEC encapsulating the request, as well as a request context that is provided by the Aggregate Manager and contains local aggregate state relevant to the request.

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

```
 sfatables → A INCOMING --
    -m hm --user-hm-prefix plc.eu.inria.thierryP --
    -j ACCEPT

 sfatables → A INCOMING --
    -m hm --user-hm-prefix plc.eu --
    -j SLICE_COUNT = e node
    -max-nodes 150
```

Each rule has three segments separated by the "−−" token: a command ("−A" for addr), which includes a chain ("INCOMING"), a match ("hm"), 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:

```
sfatables → A INCOMING --
    -m hm --user-hm-prefix plc.eu --
    -j SLICE_COUNT = e link
    -max-links 150
```

B. Examples

This section presents some of the policies that motivated this work, and in the process, explains the use of sfatables from the perspective of an aggregate's administration. Our work is relevant to a broad class of resource-providing aggregates, including testbeds such as PlanetLab and VINI, as well as cloud computing farms such as Amazon EC3. As providers make their resources available across organizational boundaries, the need arises to police these resources at a fine grain; our goal with sfatables is to make policies extensible depending on the changing needs of resource owners.

1) Node Policies

When new peers join the federation, they are typically bootstrapped with a barrier to prevent their resources from being monopolized by users of the public PlanetLab. For example, the following rule restricts users from PlanetLab Europe (PLE) to a maximum of 20 nodes per slice on PlanetLab Japan (PLJ) at the time the latter entered into the federation.

```
sfatables → A INCOMING --
    -m hm --slice-hm-prefix plc.eu --
    -j RESTRICT-SLICE-COUNT = e node
    -max-nodes 20
```

PlanetLab has a rule that prevents abuse-sensitive experiments such as ones involving peer-to-peer traffic to be bound to abuse-intolerant sites.

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

The following rule sets the maximum number of nodes used by all users on a site to 1000

```
sfatables → A INCOMING --
    -m user --admin --user-role admin --
    -j RESTRICT-SLICE-NODE-COUNT = e node
    -max-nodes 1000
```
2) Link Policies

In 2009, the VINI testbed federated with PlanetLab, giving all PlanetLab users access to the VINI substrate. One of the key resources allocated by VINI is link bandwidth, and so it was decided to limit federation users to low-bandwidth (1Mbps) virtual links in their VINI slices by default. Experiments that require more bandwidth can negotiate a higher limit with VINI administrators; for example, slice plc.imperial.ukairo has a limit of 10Mbps.

```
sfatables -A INCOMING -m slice --chk plc.imperial.ukairo -j RESTRICT_SLICE_PROP -s link -m max-link-kbps 10000
```

```
sfatables -A INCOMING -m slice --chk plc.imperial.ukairo -j ACCEPT
```

```
sfatables -A INCOMING -m all -j RESTRICT_SLICE_PROP -s link -m max-link-kbps 10000
```

3) Flow Policies

Continuing with our case that sfatables can be applied to a variety of distributed system resources, we present an example involving OpenFlow. In addition to restricting the visibility of the OpenFlow switches, the administration at Stanford required that PlanetLab users be restricted to a certain flow space, as a way of preventing them from monopolizing the networks connected to the OpenFlow switches. First, we define a rule to prevent a given user from grabbing the entire flowspec.

```
sfatables -A INCOMING -m pattern --flow -f flowspec */*/*/ */ */ */ */ */ */ * -j REJECT
```

Many PlanetLab nodes depend on DHCP for configuring their network interfaces, so users should not be allowed to hijack DHCP. 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.”

```
sfatables -A INCOMING -m pattern --flow -f flowspec */_/17/67:68/ */67:68/ */ */ */ _ _ -j REJECT
```

Having presented some actual policies that we have deployed and used as case studies for our work, we now discuss how sfatables may apply to cloud-computing substrates.

4) Donating Excess Resources

Eucalyptus [17] is an open-source cloud computing platform that is interface-compatible with Amazon’s EC2. We are currently building an Aggregate Manager for Eucalyptus that will enable Eucalyptus clusters to federate with PlanetLab. We expect that this Aggregate Manager could also be used to federate EC2 with PlanetLab.

Suppose a Eucalyptus administrator (or Amazon) wanted to contribute a fraction of their cloud’s unused computing resources to a research testbed such as GENI. The following sequence of rules might apply. The first ensures that the facility is only accessible by universities participating in GENI:

```
sfatables -A INCOMING -m site --eqgen -j whitelist geni-sites.list -- -j REJECT
```

A second rule ensures that a single slice is limited to a maximum of 10 cores.

```
sfatables -A INCOMING -m all -j LIMIT-SLICE-COUNT -s core -m max-cores 10
```

A third rule limits the total number of cores from the research network to 20% of the unallocated capacity on the cloud. Combined with the above rule, this means a single slice could allocate the maximum of 10 cores or 20% of the unallocated cores.

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

IV. DESIGN AND IMPLEMENTATION

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

- Administrators use the `frontend` to add sfatables policy rules. 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`. Policy programs are composed in XSLT, a standard language for transforming XML documents.
- The XPath query interface lets policy programs access information needed to make policy decisions. XPath is a query language for XML documents. This information includes properties of the incoming request, such as the HRN of the calling user as well as administrative information such as access control lists.
The policy compiler takes as input a set of rules and compiles them into an XSLT policy program, which we call the global policy. Since the rules that make a policy are independent, the process of compilation simply combines the rule function below into a pipeline in XSLT. Additionally, it associates this pipeline with the administrator-provided context for the match or target (e.g., “max-links” in earlier examples); we’ll discuss contexts more in Section IV-B.

```
global-rule(match, target, admin-ctx) :=
    function (request-ctx, rspec) :=
        if (match(admin-ctx, request-ctx, rspec))
            target(admin-ctx, request-ctx, rspec)
        else
            rspec
```

The matches and targets mentioned here are drawn from a library of functions implemented in XSLT. sfatables comes with a standard library of such functions. For instance, consider the following match, which checks if the requesting user is in an administrator-provided whitelist.

```
let whitelist://admin-ctx/whitelisted-users/hrn
let current_user_hrn://request-ctx/user/hrn

function recurse_list lst =
    if (length($lst) == 0) then False
    else
        let head = $lst[1]
        if starts_with($current_user_hrn, $head) then True
        else
            # XPath expr that means tail($lst) ->
            recurse_list $lst/following-sibling::*
```

```
function main =
    recurse_list($whitelist)
```

The last detail needed for the policy program in XSLT to run is a context provides concrete values for the limits imposed by the code. To this end, the sfatables tool converts the command-line parameters passed to it into XML and inserts them in the compiled program. For example, user-hrn ple.inria.thierry is converted to the following fragment.

```
<argument>
  <name>user-hrn</name>
  <value>ple.inria.thierry</value>
</argument>
```

B. Data Contexts

A context represents a collection of data values that can be read by a policy program. 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.
Syntactically, XPath expressions are like file paths in UNIX, with some additional features. In the whitelisting example above, there were two XPath expressions:

```
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 `admin-ctx` containing administrative information, and a `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 first XPath expression, and the second returns the HRN of the requesting user.

C. Runtime

To execute a given rule at runtime, `sfatables` first constructs the request context. The request context required by each match and target is declared as part of its specification using XPath expressions.

A request policy is instantiated using the values in the request context as follows:

```
request-policy := global-policy (request-ctx)
```

Next, it is applied to the incoming or outgoing RSpec:

```
output-rspec := request-policy (input-rspec)
```

V. POLICY ANALYSIS

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 feeding the constraints specified by the compiled policy into a constraint solver, such as Prolog.

A. 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.

a) 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-close 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.

b) Policy Validation.

A more automated way of validating a policy is to translate it into a Prolog knowledge base. The availability of the complete policy as a policy document makes it convenient to extract the constraints that apply to requests.

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:

```
compute_node_count (Rspec ,W),
compute_total_bw (Rspec ,X),
site_count (Rspec ,[1,2], Y),
site_count (Rspec ,[3,4], Z),
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 IV 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.
c) Policy Composition.

It may sometimes be desirable to take the policies of two aggregates and compose them. 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. Policy composition is a work-in-progress item in the implementation of sfatables.

VI. 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 [18] to global network testbeds [11] to clouds [2], [12], [7], [25]. 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 from non-federated systems (e.g., EC2 [2] and GoGrid [12]), or they can be designed explicitly to support interoperability in multiple independent systems (e.g., DeltaCloud [7] and Reservoir [25]). We adopt the SFA interfaces that were influenced by PlanetLab, but generalized for the full diversity of resources 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, 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.

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 [13], 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 embed resource allocation assertions in credentials.

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 V, 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 [27]. 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.

Another issue is how a system moderates users’ appetites for resources. This sometimes motivates a market for buying and selling resources, in which rights to re-
sources are managed by third-party brokers [9], [3], [15]. 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 sftables 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.

Grid computing resource allocation systems are related to our system in their policies and how these policies are expressed. Wu et al. [26] introduced an extensible markup language for expressing security policies in the grid, building on existing authentication systems as well as adding a policy engine to impose simple allocation policies such as time quotas for allocations. More extensive requirements for resource allocation policies in the grid have also been examined [14], [15]. Other policy frameworks for the Grid focus more on security policies rather than on resource allocation policies. Our system sftables captures a wide range of requirements in real federated distributed systems, and also has a down-to-earth implementation similar to network firewalls that is familiar to network administrators.

VII. 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. Administrators specify policies as firewall rules. These rules are compiled into XSLT documents and executed for every request. XSLT’s declarative nature makes it amenable to analysis, such as for policy composition and validation.

REFERENCES